# Metolious

**ACPI/Manageability**

*Specification V2.0 Draft*
*Document Revision 0.3*
August 15, 1999

**DRAFT SPECIFICATION**
**LICENSE AGREEMENT AND DISCLAIMERS**
**METOLIOUS ACPI/MANAGEABILITY SPECIFICATION V2.0 DRAFT**

**Intel Corporation** ("Intel") will allow you to copy this intermediate draft of the **Metolious ACPI/Manageability Specification V2.0 Draft** (the "Draft Specification") found at the URL http://developer.intel.com/design/servers/ipmi/metolious (the "Site") on the condition that you accept the terms and conditions below ("Agreement").

**IMPORTANT - READ BEFORE DOWNLOADING OR COPYING. BY SELECTING THE "I ACCEPT" BUTTON BELOW, OR BY DOWNLOADING OR COPYING THE DRAFT SPECIFICATION, YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS STATED IN THIS AGREEMENT. IF YOU SELECT "I DO NOT ACCEPT," THE DOWNLOAD PROCESS WILL NOT PROCEED. DO NOT SELECT "I ACCEPT," DOWNLOAD OR COPY THIS DRAFT SPECIFICATION UNTIL YOU HAVE CAREFULLY READ, UNDERSTOOD AND AGREED TO THE FOLLOWING TERMS AND CONDITIONS. IF YOU DO NOT WISH TO AGREE TO THESE TERMS AND CONDITIONS DO NOT DOWNLOAD OR COPY THE DRAFT SPECIFICATION.**

**NOTICE: THIS IS AN INTERMEDIATE DRAFT OF THE DRAFT SPECIFICATION PROVIDED FOR INDUSTRY REVIEW AND COMMENT ONLY. IT IS SUBJECT TO CHANGE WITHOUT NOTICE AND IS NOT THE FINAL RELEASE VERSION.** Readers must not design products based on this document.

**LICENSE.** Intel grants you a copyright license to download the Draft Specification from the Site and make copies of the Draft Specification subject to these conditions:

1. You may not copy, modify, reproduce, disclose, rent, sell, distribute, transmit or transfer all or any part of the Draft Specification except as provided in this Agreement, and you agree to use reasonable efforts to prevent such actions for any copy of the Draft Specification that you have received subject to this Agreement.
2. You may not transfer, transmit, distribute, publish, or publicly display the Draft Specification in whole or in part without the express written permission of Intel.
3. You may make copies of the Draft Specification for your own internal use for the purpose of review and providing comments or suggestions to Intel. This "Draft Specification License Agreement and Disclaimers" must be included with and all copies of this Draft Specification you make.

**NO OTHER LICENSE.** Implementations developed using the information provided in the Draft Specification may infringe the intellectual property rights of various parties including Intel. Except as expressly set forth in this Agreement, no license or right is granted to you, by implication, estoppel, or otherwise, under any patents, copyrights, maskworks, trade secrets, or other intellectual property by virtue of entering into this Agreement, downloading the Draft Specification, using the Draft Specification or building products complying with the Draft Specification.

**OWNERSHIP OF DRAFT SPECIFICATION AND COPYRIGHTS.** Title to all copies of the Draft Specification remains with Intel. The Draft Specification is copyrighted and is protected by the laws of the United States and other countries, and international treaty provisions. You may not remove any copyright or other proprietary rights notices from the Draft Specification. Intel may make changes to the Draft Specification, or to items referenced therein, at any time without notice. Intel is not obligated to support or update the Draft Specification.

**EXCLUSION OF WARRANTIES. THE DRAFT SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING WITHOUT LIMITATION ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY ARISING OUT OF ANY PROPOSAL, DRAFT SPECIFICATION OR SAMPLE. Intel disclaims all liability, direct or indirect, for any claim relating to the Draft Specification or the use of information therein including without limitation claims arising from product liability, personal injury, death, or infringement of any proprietary rights.**

**THE DRAFT SPECIFICATION IS NOT LICENSED FOR USE BY, OR INTENDED TO DIRECT OR INSTRUCT, ANY PARTY IN THE DEVELOPMENT OF ANY IMPLEMENTATION WHERE FAILURE OF THE IMPLEMENTATION COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.**

**IN NO EVENT WILL INTEL BE LIABLE FOR ANY LOSS OF PROFITS, LOSS OF USE, DIRECT, INCIDENTAL, CONSEQUENTIAL, OR SPECIAL DAMAGES, REGARDLESS OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.**

Nothing in this Agreement shall be construed as a sale or an offer for sale or license of any product.

**TERMINATION OF THIS AGREEMENT.** You may terminate this Agreement at any time upon written notice. If you breach this Agreement, Intel may terminate this Agreement at any time upon written notice. Upon termination, you will immediately destroy the Draft Specification or return all copies of the Draft Specification to the Administrator and certify in writing to the Administrator that all your copies of the Draft Specification have been returned or destroyed.

**APPLICABLE LAWS.** Claims arising under this Agreement shall be governed by the laws of Delaware, without regard to principles of conflict of laws. You may not export the Draft Specification in violation of applicable export laws and regulations.

**GOVERNMENT RESTRICTED RIGHTS.** The Draft Specification and documentation are provided with "RESTRICTED RIGHTS." Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013, et seq. Use of the Draft Specification by the Government constitutes acknowledgment of Intel's proprietary rights in them. Contractor or manufacturer is Intel as identified on the Site.

Copyright © Intel Corporation 1999, 2000
*Other brands and names are the property of their respective owners.
[Metolious ACPI/Manageability Specification V2.0 Draft on-line Draft license]

# Document Revision History

| Revision | Date | Author | Reason for Changes |
|---|---|---|---|
| Ver. 1.0 | 04/30/1999 | Intel/MCG | Initial release. Addresses mobile, desktop, and workstations |
| Ver 2.0 Draft Doc Rev 0.3 | 8/15/1999 | Intel/ESG/MCG | Included extensions to cover servers systems, including systems with IPMI-based platform management hardware. |

# Table of Contents

# List of Tables

# List of Figures

# 1 Introduction

The Advanced Configuration and Power Interface (ACPI) Specification from Intel, Microsoft and Toshiba, defines the interfaces between the operating system (OS) software, the hardware and BIOS software. The ACPI Specification allows the OS direct control over the power management and Plug and Play functions of a computer. Functional areas currently covered by the ACPI Specification include system, device, and processor power management; Plug and Play; system events; battery management; thermal management; embedded controller (EC); and system management bus (SMBus) controller access. This specification extends ACPI to include a new *manageability* functional area. This new ACPI functionality includes methods for the enumeration, access, and control of platform manageability hardware.

As one of the goals of the ACPI Specification is to facilitate, accelerate, and enhance industry-wide implementations of power management, a goal of this specification is to do the same for platform manageability. This reduces the amount of redundant investment in manageability software/firmware throughout the industry and gives OEMs more flexibility and opportunity for innovation in their designs.

This specification is focused on an efficient, robust, ACPI-based management interface that is targeted to meet the platform management requirements of desktop, mobile, workstation and server systems. This specification describes two Metolious interfaces – A basic interface targeted towards mobile, desktop, and workstation platforms and an extended interface targeted towards server platforms. While the basic interface presented in Chapters 2, 3 could also be used for entry classes of servers, the extended interface presented in Chapters 4, 5 provide a sever -specific solution to support additional scalability, extensibility, and OEM differentiation needs of high-availability entry through enterprise-class servers. The extended interface also supports server-level platform management hardware interfaces, such as the Intelligent Platform Management Interface (IPMI) specification.

## 1.1 Target Audience

This specification is intended for use by the following audience:

- OEMs who will be building ACPI-compatible systems.
- ACPI-compatible BIOS and firmware authors.
- Programmers building software solutions for manageability hardware.

Basic understanding of ACPI and platform manageability is assumed.

## 1.2 Related Documents


[1] *Advanced Configuration and Power Interface Specification*, ©1996, 1997, 1998 Intel Corporation, Microsoft Corporation, Toshiba Corporation. This specification and other ACPI documentation are available at: http://www.teleport.com/~acpi/

[2] *System Management Bus Specification*, ©1998 SBS-Implementers Forum. Available at: http://www.sbs-forum.org/smbus/

[3] *System Management BIOS Reference Specification*, © 1997, 1998 American Megatrends Inc., Award Software International, Compaq Computer Corporation, Dell Computer Corporation, Hewlett-Packard Company, Intel Corporation, International Business Machines Corporation, Phoenix Technologies Limited, and SystemSoft Corporation. Available at: http://developer.intel.com/ial/WfM/design/smbios/

[4] *Intelligent Platform Management Interface Specification v1.0*, ©1998 Intel Corporation. Available at: http://developer.intel.com/design/servers/ipmi/

[5] *Platform Management FRU Interface Storage Definition v1.0*, ©1998 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and DELL Computer Corporation.

# 1.3 Data Conventions

## 1.3.1 Data Format

All numbers specified in this document are in decimal format unless otherwise indicated. A number preceded by '0x' indicates hexadecimal format, and a number followed by the letter 'b' indicates binary format. For example, the numbers 10, 0x0A, and 1010b are equivalent.

## 1.3.2 Data Alignment

To avoid translation issues related to system endian-ness, this specification defines all values in byte-sized elements. For example, a WORD value is defined as an array of two bytes ('BYTE[2]') and a DWORD value is defined as an array of four bytes ('BYTE[4]'). All structures defined in this specification assume single-byte alignment [e.g. `#pragma pack(1)`].

### *1.3.2.1 WORD and 16-bit Integer Values*

Translation of a two-element byte array into the appropriate bit position in a WORD or 16-bit integer value is shown below.

| BYTE 0 | BYTE 1 |
|---|---|
| WORD Bits 15:8 | WORD Bits 7:0 |

### *1.3.2.2 DWORD and 32-bit Integer Values*

Translation of a four-element byte array into the appropriate bit position in a DWORD value is shown below.

| BYTE 0 | BYTE 1 | BYTE 2 | BYTE 3 |
|---|---|---|---|
| DWORD bits 31:24 | DWORD bits 23:16 | DWORD bits 15:8 | DWORD bits 7:0 |

# 1.4 Terminology

| Acronym | Description |
|---|---|
| ACPI | Advanced Configuration and Power Interface. See http://www.teleport.com/~acpi/. |
| AML | ACPI Machine Language. |
| ASIC | Application-Specific Integrated Circuit. |
| ASL | ACPI Source Language. |
| BIOS | Basic Input/Output System. |
| DMI | Desktop Management Interface. See http://www.dmtf.org/spec/dmis.html. |
| EC | Embedded Controller. |
| IPMI | Intelligent Platform Management Interface. |
| LAN | Local Area Network. |
| OEM | Original Equipment Manufacturer. |
| OS | Operating System. |
| SCI | System Control Interrupt. |
| SMBus | System Management Bus. See http://www.sbs-forum.org/smbus/. |
| WBEM | Web-Based Enterprise Management. See http://www.dmtf.org/wbem/. |

# 2 Platform Management – Basic Interface

Solution for Mobile, Desktop and Workstation platforms

## 2.1 Overview

By defining *manageability objects* in ACPI, complex tasks such as communicating over manageability buses, enumerating and controlling various manageability hardware devices, and handling peculiarities related to specific platform configurations are abstracted from manageability software. These definitions form a well-known interface allowing the capabilities of manageability hardware to be easily utilized, regardless of hardware origin or implementation. They also provide a means for the integration of ACPI and manageability, enabling coordinated access and control of manageability hardware.

## 2.2 Goals

The goals of this specification are to:

- Facilitate, accelerate, and enhance industry-wide implementations of manageability hardware capabilities.
- Provide methods for the basic integration of ACPI and manageability, allowing for coordinated access and control of shared hardware devices, and presenting solutions to a number of existing issues.
- Abstract the complexities associated with the enumeration, access, and control of manageability hardware capabilities from consumer software, allowing these capabilities to be easily utilized while providing increased flexibility and opportunity for innovation for system OEMs.

## 2.3 Problem Statement

Problems associated with legacy methods of accessing platform manageability hardware include:

- **ACPI/Manageability Coordination Issues:** Legacy manageability solutions do not coordinate the use of shared resources and control of shared devices with ACPI. This results in the potential for a number of problems, including:
  - *Invalid alerts:* When ACPI turns off a fan used in a thermal zone the manageability ASIC generates a fan low-speed RPM alert.
  - *Inadvertent changes to system policies:* Manageability software modifies a thermal sensor threshold that is also used by ACPI to implement a thermal zone.
  - *Unwanted system resets:* A system hang watchdog timer expires whenever the system enters a low-power state that prohibits a software client from resetting the timer.
  - *Resource contention:* Legacy manageability software uses a ring 0 SMBus driver and is unaware of ACPI's use of the same bus registers, resulting in (best case) data corruption or (worst case) a complete system lockup.
- **Inherent Complexity of Manageability Software**: Since current BIOS/firmware provides little or no abstraction for manageability hardware, manageability software must assume the complexity for communicating over manageability buses and controlling manageability devices.
- **Proprietary Solutions**: Complexities in utilizing manageability hardware have resulted in proprietary, OS- and platform-specific manageability software solutions.
- **Difficult to Enumerate**: Since current manageability hardware buses/devices provide no means for Plug and Play enumeration by the OS, manageability software is required to have prior knowledge of the specific platform configuration.

- **Limited Flexibility**: OEMs are given limited flexibility and room for innovation in their manageability hardware designs, as the tasks of integrating changes into legacy manageability software solutions are often prohibitive.

## 2.4 Solution

The solution defined in this specification addresses the above problems by defining methods for manageability hardware to be enumerated, accessed, and controlled through ACPI, regardless of hardware origin or implementation. The complexities of communicating over manageability buses, enumerating and controlling various manageability devices, and handling peculiarities related to specific platform configurations are abstracted through the definition of *manageability objects* in ACPI.

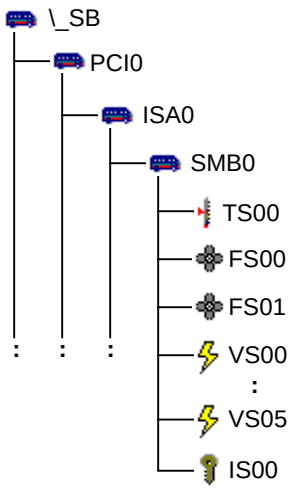
ACPI manageability objects represent the underlying capabilities of manageability hardware. For example, the Heceta II ASIC provides a thermal sensor, two fan sensors, six voltage sensors, and a chassis intrusion sensor. A separate manageability object in ACPI would represent each of these sensors.

This specification defines the following classifications of manageability objects:

- **Sensor Objects**: Thermal, fan, voltage, chassis intrusion and other sensors that relay information concerning the current environmental or other conditions within a given system.
- **Watchdog Objects**: Devices that monitor various elements of system health through the use of hardware-based timers.

As with other devices, manageability objects are added to the ACPI namespace in a method that represents the functional hierarchy of the system – meaning that the discrete manageability functions must appear as child devices of the bus they are connected to. For example, the sensors contained within a Heceta II ASIC that is connected to the system via a PIIX4 SMBus controller could be represented as shown in Figure 1.

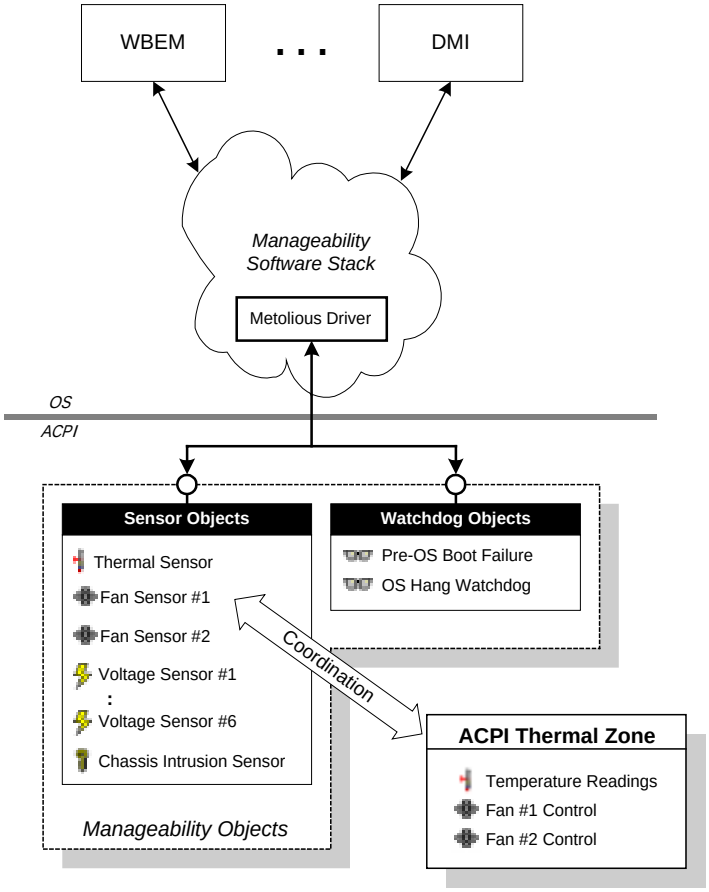
**Figure 1:** Example ACPI Namespace Hierarchy




Each manageability object is assigned a Plug and Play device identifier, resulting in the ability of the OS (and thus associated manageability drivers) to perform Plug and Play enumeration for these hardware elements. Each class includes a set of well-known control methods, providing a standard interface for manageability software drivers.

Figure 2 gives a high-level view of the manageability environment resulting from this specification for an example system. In this environment, the integration of manageability abstractions into ACPI facilitates the coordinated use of manageability hardware, while a Metolious-specific driver surfaces manageability data in a hardware-independent fashion by allowing upper level software to enumerate, access, and control manageability hardware through their manageability object definitions.

**Figure 2:** Metolious Architecture




In summary, using ACPI to define manageability hardware abstractions has the following benefits:

- Presents an OS-independent solution.
- ACPI Machine Language (AML) provides flexibility and extensibility in defining manageability hardware functionality.
- Allows coordination between ACPI and manageability, facilitating solutions to a number of potential issues.
- Isolates all platform-specific configuration in AML. The OEM is completely responsible for making manageability hardware functions visible/usable by the system, but is also given the flexibility to innovate.
- Future manageability software would not need prior knowledge about the platform to utilize the capabilities of manageability hardware.

# 3 Manageability Objects for Basic Interface

Solution for Mobile, Desktop and Workstation platforms

As described in the previous chapter, the integration of manageability hardware abstractions into ACPI is fundamental to achieving the level of coordination required to have a well-managed PC platform. This chapter provides a deeper view of this integration and the solutions proposed by this specification.

## 3.1 Overview

Examples of ACPI-Manageability integration include:

- The control of fans through a thermal zone would temporarily disable alerts for the associated fan sensor object by calling one of its control methods prior to turning off a fan. The opposite would be done when the fan is turned on by the system's thermal policy.
- A thermal sensor definition would disallow setting thresholds that may affect a thermal zone, preventing manageability software from inadvertently changing the system's thermal policy.
- A system hang watchdog would be disabled before entering a low-power state that would otherwise prohibit a client element from intermittently resetting the watchdog timer.
- Access to registers on a shared resource would be synchronized using ACPI primitives. For example, a set of control methods could be defined for communicating to a PIIX4 SMBus that include a mutex object to serialize access to the SMBus registers.

## 3.2 Manageability Objects

Manageability objects represent the underlying capabilities of manageability hardware. Access and control of a manageability device (such as a sensor or watchdog) is fully encapsulated by the device's manageability object abstraction. This modular approach is important for the following reasons:

- Facilitates the coordinated use and control of the device by multiple consumers. For example, a thermal zone and manageability software may both need to access a bus or sensor. See sections 3.3, 3.4, and 3.5.
- Allows dynamic insertion and removal of manageability hardware. For example, a mobile system may be able to access additional sensors and watchdogs when docked. See section 3.8.
- Enables Plug and Play enumeration of manageability objects by the operating system and associated driver(s).

Table 1 lists the types of manageability objects for the basic interface. Note that the range 0x01-0x7F is reserved for standard manageability objects, and the range 0x80-0xFF is reserved for vendor-defined manageability objects. This approach, which can be seen throughout this specification, is intended to encourage the use of standard definitions while allowing OEMs to develop new and innovative manageability solutions.

**Table 1:** Manageability Object Types -Basic Interface

| Value | Hardware ID | Manageability Object Type | Description | ID |
|---|---|---|---|---|
| 0x00 | MGMT0XX | <Reserved> | | |
| 0x01 | MGMT1XX | Sensor Object | Thermal, fan, voltage, chassis intrusion and other sensors that relay information concerning current environmental or other conditions within a system. | S |
| 0x02 | MGMT2XX | Watchdog Object | Devices that monitor various elements of system health through the use of hardware-based timers. | W |
| 0x80+ | MGMT8XX+ | <Vendor-Defined Manageability Objects> | | |

The following ACPI Source Language (ASL) demonstrates how thermal sensor hardware is encapsulated through the definition of a sensor object. This example is based on the ACPI namespace shown in Figure 1. Note that the temperature is read by executing the SR (sensor reading) control method from the thermal sensor object (TS00). All access and control of this thermal sensor will be performed using the interface exposed by the TS00 object.

```
Scope(\_TZ)
{
    :
    // Create a thermal zone.
    ThermalZone(THRM) {
        :
        // Get the current temperature.
        Method(_TMP, 0) {
            // Read the temperature (in 1/10 °C) from the thermal sensor object
            // (versus reading directly from the hardware) and convert to
            // 1/10°K (Kelvin = Celsius + 273.2).
            Store(\_SB.PCI0.ISA0.SMB0.TS00.SR, Local0)
            Add(Local0, 2732, Local0)
            Return(Local0)
        }
        :
    }
    :
}

Scope(\_SB)
{
    :
    Device(PCI0) {
        :
        Device(ISA0) {
            :
            // PIIX4 SMBus device.
            Device(SMB0) {
                :
                // Thermal Sensor #1.
                Device(TS00) {
                    // Sensor Information: Return SENSOR_INFO structure.
                    Method(INFO, 0) {…}
                    // Sensor Reading: Return current temperature in 1/10 °C.
                    Method(SR, 0) {…}
                    // Sensor State: Return state based on current thresholds.
                    Method(SS, 0) {…}
                }
            }
        }
    }
}
```

Complete definitions for sensor and watchdog objects are provided in sections 3.10 and 3.15, respectively.

## 3.3 Synchronizing Access

The need to synchronize access to shared resources is often required when the potential for multiple simultaneous consumers exists. This is the case with manageability hardware in ACPI environments, as it is common for ACPI, manageability software, and system-level firmware to need access to the same physical device. For example, the SMBus may be used by ACPI to access thermal sensor readings for a thermal zone, by manageability software to communicate to various sensors, and by firmware to perform various battery functions.

Due to this environment, care should be taken to ensure that access to shared devices is serialized. This typically involves replacing drivers that directly access manageability hardware directly with ACPI-friendly alternatives and through the prudent use of the global lock.

Note that the Smart Battery System Implementers Forum addresses issues related to SMBus access in an ACPI environment in their *SMBus Control Method Interface (CMI) Specification*. In fact, the sample ASL (for SMBus access) presented in Appendix A is based upon the version of the SMBus CMI Specification available at the time of this specification's release. ACPI BIOS developers should obtain the latest version of the SMBus CMI Specification, which is available at:

http://www.sbs-forum.org/smbus/

## 3.4 Coordinating Thermal Management

The need to coordinate access and control of manageability objects within the context of an ACPI thermal policy stems from the fact that thermal zones and manageability software often reference the same hardware. ACPI developers need to be cognizant of how sensors (and the devices being monitored by sensors) are affected by the thermal policy in order to produce a well-managed system. Recommendations include:

- *Encapsulate thermal and fan sensor access into their appropriate manageability object definitions.* Thermal zones should access sensors through their sensor object definitions (not directly). For example, to read the current temperature a thermal zone should call the appropriate thermal sensor object's `SR()` control method. This policy helps ensure synchronized access to shared resources.
- *Disallow modifications to thermal sensor thresholds from manageability software to prevent inadvertent changes to the thermal policy.* When a thermal sensor is used in a thermal zone its thresholds (trip points) are often used to generate a thermal SCI to notify ACPI of changes in the processor temperature. The OS in turn is then able to apply the appropriate cooling policy based on the current temperature. Modification of these thresholds by manageability software would undermine the platform's thermal policy.
- *Update a fan sensor's state in relation to changes to the fan speed.* For example, ACPI should disable thresholds on a fan sensor prior to turning the fan monitored by the sensor off. Fans supporting multiple speeds would require finer control of sensor thresholds.

## 3.5 Coordinating Power Management

In a manner similar to thermal management, ACPI developers need to be cognizant of how manageability objects are affected by changes to the power state. This (again) is rooted in the fact that changes made through ACPI often have a direct or indirect affect on manageability hardware. Recommendations include:

- *Update a power quality sensor's state in relation to changes to the power state.* For example, ACPI should disable thresholds on a voltage sensor prior to turning off the power plane that the sensor is monitoring.
- *Update a device presence sensor's state in relation to the monitored device's power state.* For example, a sensor that detects the absence of a processor may need to be disabled prior to turning off power to the processor. Failure to do so may result in the generation of an invalid remote alert.
- *Enable or disable watchdogs based on the power state.* For example, ACPI should disable a system hang watchdog before entering a low-power state that would otherwise prohibit the watchdog client from intermittently resetting the watchdog timer.

## 3.6 Hardware Alerting

The ability of manageability hardware to asynchronously notify consumers of a change in its state is referred to by this specification as *hardware alerting*. This highly desirable capability relieves upper-level software from implementing poll-based policies and generally results in a much more responsive and accurate environment.

### 3.6.1 Local Alerting

Local hardware alerting is defined by this specification as the ability for manageability hardware to generate an ACPI-visible notification whenever a change in the hardware's state is detected. These notifications are the basis for upper level software to generate DMI or WBEM events.

The ACPI-defined mechanism for generating asynchronous notifications is through a system control interrupt (SCI). An SCI occurs whenever one or more non-masked bits are set in the ACPI event status register. Each bit in the event status register has associated AML that serve as the SCI interrupt handler. It is the job of this AML to determine the root cause of the SCI and perform the required action. See the ACPI specification for more information.

The AML code that handles an SCI for a manageability event notifies the system of a change in the state of a sensor or watchdog object. After decoding the root cause of the event, the AML must issue a `Notify(0x80)` command targeting the associated sensor or watchdog object. For example, if a single event status register were used to represent the hardware alerting capability of a Maxim 1617* ASIC, the interrupt handler AML would need to read the status byte from the ASIC to determine which sensor (local or remote) caused the alert. Assuming that the local (ambient) temperate sensor was the culprit, and the associated sensor object existed in the namespace as `"TS01"` within the context of an EC-SMBus, the AML would then issue the command:

```
Notify(\_SB.EC0.SMB0.TS01, 0x80)
```

### 3.6.2 Remote Alerting

Remote hardware alerting is defined by this specification as the ability for manageability hardware to directly forward an alert to a remote consumer over a LAN, alphanumeric paging, or other data communication medium. This capability allows manageability notifications to occur even when critical components of the system (such as the operating system) are non-functional.

## 3.7 Hardware Initialization

Manageability hardware often needs to be initialized to enable data collection, set default thresholds, or perform other preparatory tasks. In a broad sense, hardware can be initialized either by the BIOS (immediately after system power-on) or by ACPI (during OS start-up).

### 3.7.1 Via the BIOS

Certain manageability hardware needs to be operational as soon after initial power-on as possible. In these cases the BIOS must perform the required actions. For example, the hardware implementing a BIOS boot failure watchdog may not be operational until a configuration register is initialized. (An example of initializing a LM75 ASIC via BIOS is provided in section 8.1.3 of the *ACPI Implementers' Guide*.)

### 3.7.2 Via ACPI

ACPI provides several means for initializing manageability devices that don't need to be operational immediately after power-on. The ACPI specification provides two control methods for initializing hardware: `_REG()` and `_INI()`. Additionally, this specification defines the `INIT()` control method. Details of how and when to use these control methods are discussed below.

#### *3.7.2.1 _REG()*

The `_REG()` control method is defined in section 6.5.4 of the ACPI specification. This control method is used to inform AML when a driver that controls an operation region is ready – or no longer ready – for access. This control method is useful for performing general ASIC-level initialization for devices residing on a SMBus. See the ACPI specification for more information.

#### *3.7.2.2 _INI()*

The `_INI()` control method is defined in section 6.5.1 of the ACPI specification. This control method is executed exactly once by the ACPI subsystem shortly after ACPI has been enabled. There are restrictions on using this

control method (e.g. cannot be used for devices connected to a SMBus that uses an `OperationRegion` interface). This control method, like `_REG()`, is useful for performing general ASIC-level initialization. See the ACPI specification for more information.

#### *3.7.2.3 INIT()*

The INIT() control method is defined by this specification and intended for use in performing sensor- or watchdog-specific initialization. For example, programming the upper critical threshold and RPM divisor for a fan sensor on a Heceta II ASIC would be implemented by this control method. Since the Heceta II has several sensors, it would not be prudent to perform ASIC-specific initialization using this control method.

The Metolious driver executes this control method exactly once immediately after the successful enumeration of a Metolious device. Enumeration occurs upon system startup or whenever the device is dynamically inserted into the system (e.g. hot docking).

# 3.8 Dynamic Insertion & Removal

As with other ACPI devices, when a Metolious device is inserted or removed from the system, the hardware asserts a general-purpose event. The AML code handler for this event will issue a `Notify()` command on the manageability object to initiate the standard device Plug and Play actions.

For example, a notebook system's docking station may include a thermal sensor that is defined in the namespace as `"TS01"`. During the hot dock/undock process an ACPI-visible event would be triggered by hardware. The AML handler for this event would, among other things, issue a 'device check' notification for `TS01`. This in turn causes the OS to re-evaluate this object's `_STA` control method and determine whether the device has been added or removed from the system.

See section 5.6.3 of the ACPI specification for more information.

# 3.9 Suspend/Resume

Platforms supporting robust power management may be capable of setting manageability hardware into a low power state (e.g. `D3`) when not in active use. This hardware may require persistent storage, re-initialization, or other tasks to be performed during transition to/from a low power state. The ACPI specification allows BIOS developers to manage these tasks via the `_PSx` control methods (e.g. `_PS3`). See the ACPI specification for more information.

# 3.10 Sensor Objects

Environmental conditions such as system voltages, baseboard and processor temperatures, fan speed, and chassis intrusion can provide crucial information to manageability software, helping to head off problems before they occur. Through the definition of *sensor objects* in ACPI, this specification facilitates the coordinated use of sensor hardware between ACPI and manageability, and presents an interface for making sensor information and control universally available to management software.

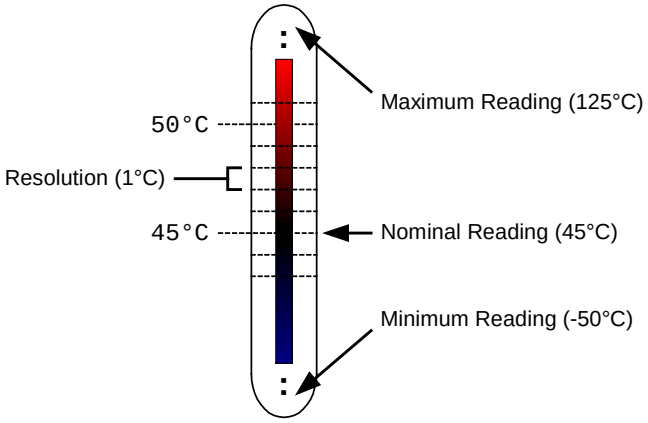
This chapter provides details on how to abstract standard sensors and presents a foundation for building vendor-defined sensor types.

A sensor is defined as any hardware-based device that monitors a PC hardware, firmware, or software component. Sensors are often categorized by the range of values produced (e.g. *numeric*) and by their ability to monitor and detect changes in their state (e.g. *state-based*).

## 3.10.1 Numeric Sensors

Numeric sensors are characterized by their ability to provide data that can span a wide range of values, usually corresponding to some analog-like property of the device being monitored. For example, a thermal sensor may sample the processor temperature every second and, using an analog-to-digital converter, present this information in an 8-bit binary format. Due to the analog-like nature of their sources, numeric sensors often have properties such as *resolution*, *tolerance*, and *maximum reading* that can be used by higher-level consumers to understand the capabilities of the sensor and the exactness of the data being represented.

**Figure 3:** Numeric Thermal Sensor




For example, the numeric thermal sensor illustrated in Figure 3 is physically capable of reading temperatures between 125°C and –50°C. The data format is 7 bits plus sign, with each bit corresponding to 1°C, in twos-compliment fashion. The device being monitored by this sensor has a normal operating temperature of 45°C.

## 3.10.2 State-Based Sensors

State-based sensors are able to provide a discrete value that represents the current state of the device being monitored. State-based sensors must support at least two (to a maximum of 32) states. *Binary Sensors* are defined as state-based sensors that support only two possible states.

State-based sensors often have the ability to originate hardware alerts upon a state transition, relieving higher-level consumers from implementing a poll-based event generation policy (see section 3.10.4).

### *3.10.2.1 Sensor States*

Table 2 lists the set of states defined for state-based sensors. A maximum of 32 states is supported. The first 16 states are known as the *standard states* – in that they are based on the states supported by DMI and WBEM. States 16 through 30 are reserved for vendor-specific purposes.

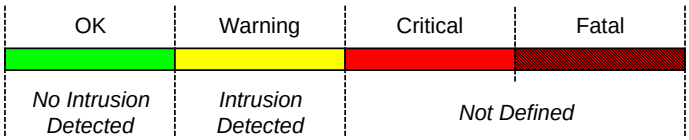
**Table 2:** Sensor States

| Value | State Name | Description |
|---|---|---|
| 0 | OK | Normal operating state. |
| 1 | Warning | Warning (non-critical) state. |
| 2 | Critical | Critical state. |
| 3 | Fatal | Fatal (non-recoverable) state. |
| 4:15 | <Reserved> | Reserved for future standard states. Should be cleared (0). |
| 16:30 | <Vendor-defined> | Vendor-defined sensor states. |
| 31 | <Reserved> | Reserved. Used to indicate an unknown state. |

The term *standard state-based sensor* implies the use of the standard states (or subset thereof). This specification defines a set of standard sensor types (listed in section 3.11) that account for most of the PC platform manageability sensors available today. OEMs are encouraged to use vendor-defined sensor types and/or states when the standard definitions fail to meet the requirements of the underlying hardware.

The chassis intrusion sensor described in section 3.11.4.1 is an example of a standard state-based sensor. It would be implemented using two states: *OK* and *Warning*. The *Warning* state implies that a chassis intrusion has been detected, while the *Critical* and *Fatal* states are not defined.

**Figure 4:** State-Based Chassis Intrusion Sensor




## 3.10.3 State-Based Numeric Sensors

State-based numeric sensors include characteristics of both numeric and state-based sensors. These sensors can be thought of as "self-realized", in that they are able to monitor and detect when their numeric reading crosses some predefined state boundary (threshold). Some sensors also support hysteresis, which helps limit the number of state transitions that may occur when sensor readings fluctuate tightly around a given threshold.

### *3.10.3.1 Thresholds*

State-based numeric sensors use thresholds to define their state boundaries. They may define upper and lower boundaries (thresholds) for a given state, or may only support certain states.

Table 3 lists the set of thresholds defined for state-based numeric sensors. A maximum of 32 thresholds is supported. The first 16 thresholds are known as the *standard thresholds* – in that they are based on the thresholds supported by DMI and WBEM. Thresholds 16 and above are reserved for vendor-specific purposes.

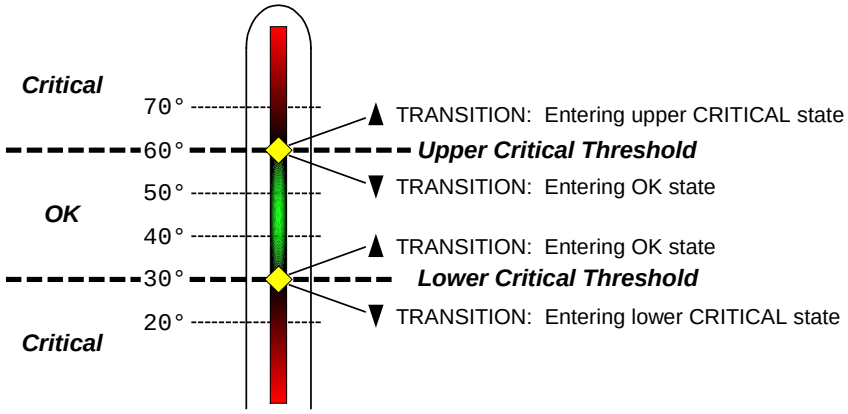
**Table 3:** Sensor Thresholds

| Value | Threshold Name | Description |
|---|---|---|
| 0 | Upper Warning | The upper warning (non-critical) threshold. |
| 1 | Lower Warning | The lower warning (non-critical) threshold. |
| 2 | Upper Critical | The upper critical threshold. |
| 3 | Lower Critical | The lower critical threshold. |
| 4 | Upper Fatal | The upper fatal (non-recoverable) threshold. |
| 5 | Lower Fatal | The lower fatal (non-recoverable) threshold. |
| 6:15 | <Reserved> | Reserved for future standard thresholds. |
| 16:31 | <Vendor-defined> | Vendor-defined sensor thresholds. |

The term *standard state-based numeric sensor* implies the use of the standard thresholds (or subset thereof). OEMs are encouraged to use vendor-defined thresholds when the standard definitions fail to meet the requirements of the underlying hardware.

For example, the standard state-based numeric thermal sensor illustrated in Figure 5 defines an upper critical threshold of 60° C, a lower critical threshold of 30° C, and no warning or fatal thresholds. In this example the *OK* state can be deduced to consist of the temperature range between 30° and 60° C – any reading outside of this range results in a *Critical* state (as the *Warning* and *Fatal* states are not defined). If hardware alerting were supported, a separate alert would be generated by this sensor during each state transition.

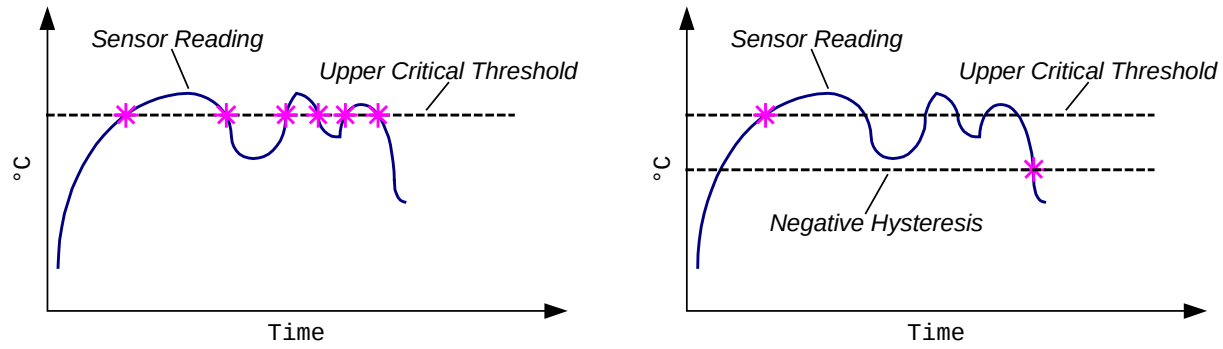
**Figure 5:** State-Based Numeric Thermal Sensor




#### *3.10.3.2 Hysteresis*

State-based numeric sensors with hardware hysteresis support can significantly reduce state transition 'storms' when sensor readings fluctuate tightly around a given threshold. For example, the readings shown in Figure 6 are for a state-based numeric thermal sensor. Without hardware hysteresis support a state transition occurs every time the temperature crosses the upper critical threshold boundary (six times). By employing hysteresis for the upper critical threshold the sensor in this example can delay changing states until the reading drops below the hysteresis-modified boundary (twice).

**Figure 6:** Hardware Hysteresis




This specification allows the assignment of a single hysteresis value that applies to all defined thresholds. The type of threshold determines whether the hysteresis is negative (less than the given threshold) or positive (greater than the given threshold): hysteresis assigned to upper thresholds (e.g. upper critical) are always negative, and hysteresis assigned to lower thresholds (e.g. lower warning) are always positive.

For example, a hysteresis value of 0x00000064h for state-based numeric thermal sensor shown in Figure 6 would indicate a hysteresis value of 10.0°C (units are in $1/10^{th}$ °C). Assuming that the upper critical threshold in this example is set to 60.0°C, the associated hysteresis would be set to 50.0°C.

Note that ACPI defines hysteresis in a slightly different context when discussing thermal management in chapter 12 of the specification. Because of this discrepancy it is recommended that hardware hysteresis not be used on thermal sensors that are also used in the definition of a thermal zone, as this may impact the thermal sensor's ability to generate hardware alerts and thus alter the thermal policy.

### 3.10.4 Hardware Alerting

Sensors supporting hardware alerts relieve higher-level consumers from implementing poll-based event generation policies, and generally facilitate a much more responsive and accurate environment. Sensors generate hardware alerts upon state transitions – and thus only state-based sensors can originate alerts. An alert must be generated upon every state transition for all supported (enabled) states.

As described in section 3.6, this specification defines local hardware alerting as the ability for manageability hardware to generate an ACPI-visible asynchronous notification (e.g. SCI) which then can be decoded to a specific sensor and then routed up the manageability software stack. Remote alerting is defined as the ability for manageability hardware to directly send an alert to a remote consumer over a LAN, alphanumeric paging, or other data communication medium.

For example, if the state-based numeric thermal sensor shown in Figure 6 supported local hardware alerting, a separate alert would be generated immediately following each state transition. Without hysteresis, six hardware alerts would be generated. Through the use of hysteresis the number of hardware alerts drops to two, as state transitions are prevented while the sensor reading fluctuates tightly around the upper critical threshold.

### 3.10.5 Monitored Device

Associating a sensor with the device or subsystem it monitors is required to enable robust management of a platform. This specification defines two methods for establishing such relations. The first method allows a sensor to describe the general type and instance information of the device being monitored, and is discussed in section 3.14.1. The second method (optional) allows a sensor to reference a SMBIOS table entry that provides additional information about the monitored device – see the *SMBIOS Table Entry Reference* property in section 3.14.2.1.

# 3.11 Type Details

This specification defines several classes of sensors, representing the most common manageability sensors currently available for PC platforms, as shown below in Table 4. OEMs with platforms using hardware sensors that are not covered by this specification are encouraged to use vendor-defined types. Details on each sensor type are provided in the remainder of this section.

**Table 4:** Standard Sensor Types

| Major Type Value | Hardware ID | Major Sensor Type | Minor Type Value | Minor Sensor Type | ID |
|---|---|---|---|---|---|
| 0x00 | MGMT100 | <Reserved> | | | |
| 0x01 | MGMT101 | Thermal | 0x00 | <Reserved> | – |
| | | | 0x01 | Standard Thermal Sensor | T |
| 0x02 | MGMT102 | Cooling Device | 0x00 | <Reserved> | – |
| | | | 0x01 | Standard Fan Sensor | F |
| 0x03 | MGMT103 | Power Quality | 0x00 | <Reserved> | – |
| | | | 0x01 | Standard Voltage Sensor | V |
| | | | 0x02 | Standard Current Sensor | C |
| 0x04 | MGMT104 | Physical Security | 0x00 | <Reserved> | – |
| | | | 0x01 | Standard Chassis Intrusion Sensor | I |
| | | | 0x02 | Standard LAN Leash Sensor | L |
| | | | 0x03 | Standard Device Presence Sensor | P |
| 0x04 – 0x7F | MGMT104 – MGMT17F | <Reserved for Future Standard Sensors> | | | |
| 0x80+ | MGMT180 – MGMT1FF | <Vendor-Defined Sensors> | | | |

## 3.11.1 Thermal Sensors

Thermal sensors are defined as any sensor that monitors the thermal characteristics of a given device or zone. Thermal sensors can be numeric or state-based. As with all standard sensors, state-based thermal sensors base their states on the standard states (or subset thereof) defined in Table 2.

Since thermal sensors are often used to implement an ACPI thermal zone, care should be taken to coordinate the sensor's use between ACPI and manageability. This primarily involves prohibiting the modification of sensor thresholds (state boundaries) from manageability consumers to avoid unintentional alterations to the system policy. Additional details on this subject are provided in section 3.4.

### *3.11.1.1 Numeric Units*

For numeric thermal sensors, the interpretation for sensor readings and other numeric properties are shown below.

| Property | Interpretation |
|---|---|
| Readings & Defaults | $1/10^{th}$ °C |
| Accuracy, Tolerance & Resolution | ± $1/10^{th}$ °C |
| Hysteresis | Unit deviation in $1/10^{th}$ °C |

### *3.11.1.2 State Interpretations*

For state-based thermal sensors, the interpretations for the standard states provided in Table 2 are shown below.

| State | Interpretation |
|---|---|
| OK | Temperature is within normal operating parameters. |
| Warning | Temperature has reached a warning (non-critical) state. |
| Critical | Temperature has reached a critical state. |
| Fatal | Temperature has reached a fatal (non-recoverable) state. |

## 3.11.2 Cooling Device Sensors

### *3.11.2.1 Fan Sensors*

Fan sensors are numeric or state-based cooling device sensors that monitor the rotational speed of fan-like cooling devices. Since fans are often used in the implementation of an ACPI thermal zone, care should be taken to coordinate ACPI control of the monitored device (fan) with manageability. This primarily involves updating fan sensor thresholds whenever ACPI changes the fan speed – thus avoiding invalid state changes. Additional details on this subject are provided in section 3.4.

#### *3.11.2.1.1 Numeric Units*

For numeric fan sensors, the interpretation for sensor readings and other numeric properties are shown below.

| Property | Interpretation |
|---|---|
| Readings & Defaults | Revolutions-Per-Minute (RPM) |
| Accuracy, Tolerance & Resolution | ± RPM |
| Hysteresis | Unit deviation in RPM |

#### *3.11.2.1.2 State Interpretations*

For state-based fan sensors, the interpretations for the standard states provided in Table 2 are shown below.

| State | Interpretation |
|---|---|
| OK | Fan RPM speed is within normal operating parameters. |
| Warning | Fan RPM speed has reached a warning (non-critical) state. |
| Critical | Fan RPM speed has reached a critical state. |
| Fatal | Fan RPM speed has reached a fatal (non-recoverable) state. |

## 3.11.3 Power Quality Sensors

Power quality sensors are defined as any sensor that monitors the characteristics of a power source. Since the power source being monitored may be under the control of ACPI as part of a power management policy, care should be taken to coordinate ACPI control of the power source with manageability. This primarily involves updating sensor thresholds whenever the OS changes the power state – thus avoiding invalid state changes. Additional details on this subject are provided in section 3.5.

### *3.11.3.1 Voltage Sensors*

Voltage sensors are numeric or state-based power quality sensors that monitor electric voltage characteristics.

#### *3.11.3.1.1 Numeric Units*

For numeric voltage sensors, the interpretation for sensor readings and other numeric properties are shown below.

| Property | Interpretation |
|---|---|
| Readings & Defaults | Millivolts |
| Accuracy, Tolerance & Resolution | ± Millivolts |

| | |
|---|---|
| Hysteresis | Unit deviation in Millivolts |

#### *3.11.3.1.2 State Interpretations*

For state-based voltage sensors, the interpretations for the standard states provided in Table 2 are shown below.

| State | Interpretation |
|---|---|
| OK | Voltage is within normal operating parameters. |
| Warning | Voltage has reached a warning (non-critical) state. |
| Critical | Voltage has reached a critical state. |
| Fatal | Voltage has reached a fatal (non-recoverable) state. |

### *3.11.3.2 Electrical Current Sensors*

Electrical current sensors are numeric or state-based power quality sensors that monitor electrical current characteristics.

#### *3.11.3.2.1 Numeric Units*

For numeric electrical current sensors, the interpretation for sensor readings and other numeric sensor properties are shown below.

| Property | Interpretation |
|---|---|
| Readings & Defaults | Milliamps |
| Accuracy, Tolerance & Resolution | ± Milliamps |
| Hysteresis | Unit deviation in Milliamps |

#### *3.11.3.2.2 State Interpretations*

For state-based electrical current sensors, the interpretations for the standard states provided in Table 2 are shown below.

| State | Interpretation |
|---|---|
| OK | Electrical current is within normal operating parameters. |
| Warning | Electrical current has reached a warning (non-critical) state. |
| Critical | Electrical current has reached a critical state. |
| Fatal | Electrical current has reached a fatal (non-recoverable) state. |

## 3.11.4 Physical Security Sensors

### *3.11.4.1 Chassis Intrusion Sensor*

Chassis intrusion sensors are non-numeric state-based physical security sensors that detect when a chassis has been opened. These sensors are typically designed to accept an active high signal from an external circuit that latches when the case is removed from the computer. The external circuit can often operate even when the system is in a low power or OFF state, remaining in an 'intruded' state until manually reset (re-armed).

#### *3.11.4.1.1 Numeric Units*

Numeric chassis intrusion sensors are not defined by this specification.

#### *3.11.4.1.2 State Interpretations*

The interpretations for the standard states provided in Table 2 are shown below.

| State | Interpretation |
|---|---|
| OK | No chassis intrusion detected. |
| Warning | Chassis intrusion detected. |
| Critical | Not defined. |
| Fatal | Not defined. |

#### *3.11.4.2 LAN Leash Sensors*

LAN leash sensors are non-numeric state-based physical security sensors that monitor the physical state of a network cable connection. These sensors conceptually view the network cable as a security device that, when disconnected, indicates the possibility that the system was physically moved.

##### *3.11.4.2.1 Numeric Units*

Numeric LAN leash sensors are not defined by this specification.

##### *3.11.4.2.2 State Interpretations*

The interpretations for the standard states provided in Table 2 are shown below.

| State | Interpretation |
|---|---|
| OK | LAN link indication present. |
| Warning | LAN link indication lost. |
| Critical | Not defined. |
| Fatal | Not defined. |

#### *3.11.4.3 Device Presence Sensor*

Device presence sensors are non-numeric state-based physical security sensors that are typically used to detect the absence of a crucial device (e.g. processor). Note that the type of device being monitored is specified in the *Monitored Device* property of the sensor object (see section 3.14.1.2).

##### *3.11.4.3.1 Numeric Units*

Numeric device presence sensors are not defined by this specification.

##### *3.11.4.3.2 State Interpretations*

The interpretations for the standard states provided in Table 2 are shown below.

| State | Interpretation |
|---|---|
| OK | Device is present. |
| Warning | Device is absent. |
| Critical | Not defined. |
| Fatal | Not defined. |

## 3.12 ASL Definition

The ACPI definition for sensor objects is provided below. Details on control methods are provided in section 3.13. Details on data structures are provided in section 3.14.

```
Device(<type>S<id>){
   Name(_HID, <hid>)          // Sensor Hardware ID (PnP ID)
   Name(_UID, <uid>)          // Sensor Signature
   Method(INIT, 0) {…}        // Sensor Hardware Initialization
   Method(INFO, 0) {…}        // Sensor Information
```

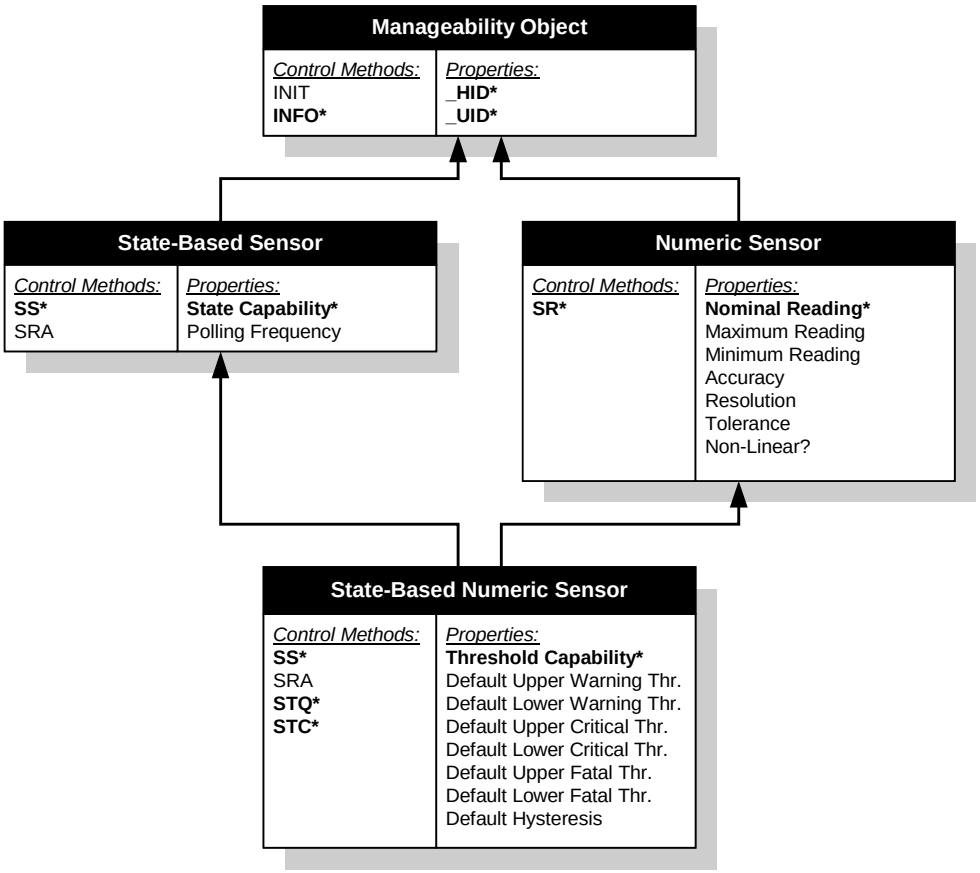
```
    Method(SR, 0) {…}          // Sensor Readings
    Method(SS, 0) {…}          // Sensor State
    Method(SRA, 0) {…}         // Sensor Re-Arm
    Method(STQ, 1) {…}         // Sensor Threshold Query
    Method(STC, 2) {…}         // Sensor Threshold Control
    Method(SHQ, 0) {…}         // Sensor Hysteresis Query
    Method(SHC, 1) {…}         // Sensor Hysteresis Control
}
```

### 3.12.1 Requirements

Figure 7 expresses the sensor classifications and associated ASL requirements using an object inheritance paradigm. Derived sensor 'classes' (e.g. *State-Based Numeric*) inherit the control method and property requirements of all parent 'classes' (e.g. *Sensor* and *Numeric*). This diagram is simply intended to illustrate the ASL requirements for sensor objects, and does not infer an object-oriented development process.

Required control methods and properties for each object type are indicated with an asterisk ('*').

**Figure 7:** Sensor Requirements Hierarchy




For example, a sensor object representing the chassis intrusion detection hardware on a Heceta II would be classified as a non-numeric state-based sensor. This sensor object must implement the `INFO()` and `SS()` control methods, as well as the *State Capability* property. Since the external circuit used to detect an intrusion typically remains asserted until it is manually cleared (re-armed), `SRA()` should be implemented.

### 3.12.2 Name

Manageability object names, as with any object in ACPI, must follow the ASL naming convention defined in section 15.1.2 of the ACPI Specification. Below is the *recommended* format for specifying the name of a sensor object.

Although any ACPI-compliant name can be used, this format allows sensors to be easily identified when inspecting ASL.

**`Device(<type>S<id>)`**

`<type>` A single character used to specify the sensor object type. This specification recommends using the *ID* field from Table 4. For example the letter `'T'` would be used for a thermal sensors.

`S` A single character that specifies the type of manageability object. This specification recommends using the value presented in the *ID* field from Table 1. The letter `'S'` is used for sensor objects.

`<id>` A two-character identifier that, together with the previous two characters, creates a device name that is unique throughout all ACPI device objects. This specification recommends using the same value as used for the *Sensor Instance* field of the `SENSOR_INFO` structure (see section 3.14.1).

### 3.12.3 Device Identification

Manageability objects utilize the _HID and _UID device identification objects to facilitate Plug and Play enumeration, providing the capability for the 'automatic' enumeration of each object class by the OS.

**`Name(_HID, "MGMT1XX")`**

Used to specify the Plug and Play hardware ID for this device. See the *Hardware ID* field from Table 4. For example, a physical security sensor would have a HID value of "MGMT104".

**`Name(_UID, <uid>)`**

`<uid>` A 32-bit value used to specify a unique identifier for this sensor object. This specification recommends using a combination of the *Object Type*, *Major Sensor Type*, *Minor Sensor Type*, and *Sensor Instance* fields of the `SENSOR_INFO` structure (see 3.14.1). For example, the first instance of a chassis intrusion sensor would have a UID value of 0x01040100.

## 3.13 Control Methods

### 3.13.1 Initialize (INIT)

| | |
|---|---|
| **Description:** | This control method is called by a supporting driver immediately after the OS enumerates a sensor object, and is used to perform any required initialization of the sensor hardware. |
| **Argument(s):** | <none> |
| **Return Value:** | <none> |
| **Notes:** | Optional. Implement only if hardware must be initialized prior to its use. |

### 3.13.2 Information (INFO)

| | | |
|---|---|---|
| **Description:** | Return a `SENSOR_INFO` structure describing the general properties of a sensor object. | |
| **Argument(s):** | <none> | |
| **Return Value:** | Sensor Information (Buffer) | A `SENSOR_INFO` structure as a byte array. See section 3.14.1. |
| **Notes:** | Required for all sensors. | |

### 3.13.3 Sensor Reading (SR)

| | | |
|---|---|---|
| **Description:** | Returns the current reading value for numeric sensors. | |
| **Argument(s):** | <none> | |
| **Return Value:** | Sensor Reading (Integer) | The current sensor reading. The interpretation of this value depends on the sensor object type. See section 3.11. Note that the value 0x80000000 indicates that the sensor reading is unknown. |
| **Notes:** | Required for all numeric and state-based numeric sensors. | |

### 3.13.4 Sensor State (SS)

| | | |
|---|---|---|
| **Description:** | Returns the current state of a sensor. | |
| **Argument(s):** | <none> | |
| **Return Value:** | State ID (Integer) | A bitmap representing the current sensor state. A set bit (1) indicates that the associated state is currently enabled, while a cleared bit (0) indicates that the state is disabled.<br>Bit 0 – OK<br>Bit 1 – Warning<br>Bit 2 – Critical<br>Bit 3 – Fatal<br>Bits 4:15 – Reserved. Should be cleared (0).<br>Bits 16:30 – Vendor-defined states.<br>Bit 31 – Reserved<br>Use the *State Capability* sensor property to determine which states are supported (see section 3.14.2.1.1). The interpretation of these states depends on the sensor type (see section 3.11). Note that the value 0x80000000 indicates that the state is unknown or currently unavailable, and the value 0x00000000 is reserved. |
| **Notes:** | Required for all state-based sensors. | |

### 3.13.5 Sensor Re-Arm (SRA)

| | |
|---|---|
| **Description:** | Re-arm a manually arming sensor. |
| **Argument(s):** | <none> |
| **Return Value:** | <none> |
| **Notes:** | Required for state-based sensors that need to be manually re-armed following a state transition. This control method must be invoked by upper-level software following the consumption of a state change on a manually arming sensor. |

### 3.13.6 Sensor Threshold Query (STQ)

| | | |
|---|---|---|
| **Description:** | Query the current value of a hardware threshold. | |
| **Argument(s):** | Threshold ID (Integer) | A bitmap that specifies which threshold to perform the operation on. Setting a bit (1) indicates that the associated threshold should be included in the operation. Only a single bit may be set.<br>Bit 0 – Upper Warning Threshold<br>Bit 1 – Lower Warning Threshold<br>Bit 2 – Upper Critical Threshold<br>Bit 3 – Lower Critical Threshold<br>Bit 4 – Upper Fatal Threshold<br>Bit 5 – Lower Fatal Threshold<br>Bits 6:15 – Reserved. Should be cleared (0).<br>Bit 16:31 – Vendor-Defined Thresholds.<br>Use the *Threshold Capability* sensor property to determine which thresholds are supported (see section 3.14.2.1.2). |
| **Return Value:** | Threshold Value (Integer) | The current threshold value. The interpretation of this value depends on the sensor object type. Note that the value 0x80000000 indicates that the threshold value is unknown or currently unavailable. The value 0x7FFFFFFF indicates that the threshold is currently disabled. |
| **Notes:** | Required for all state-based numeric sensors. | |

### 3.13.7 Sensor Threshold Control (STC)

| | | |
|---|---|---|
| **Description:** | Modify the value of a hardware threshold. | |
| **Argument(s):** | 1. Threshold ID (Integer) | A bitmap that specifies which threshold to perform the operation on. Setting a bit (1) indicates that the associated threshold should be included in the operation. Multiple bits may be set.<br>Bit 0 – Upper Warning Threshold<br>Bit 1 – Lower Warning Threshold<br>Bit 2 – Upper Critical Threshold<br>Bit 3 – Lower Critical Threshold<br>Bit 4 – Upper Fatal Threshold<br>Bit 5 – Lower Fatal Threshold<br>Bits 6:15 – Reserved. Should be cleared (0).<br>Bit 16:31 – Vendor-Defined Thresholds.<br>Use the *Threshold Capability* sensor property to determine which thresholds are supported (see section 3.14.2.1.2). |
| | 2. Threshold Value (Integer) | The new threshold value. The format for this value depends on the sensor object type. Use the value 0x7FFFFFFF to disable a threshold. |
| **Return Value:** | <none> | |
| **Notes:** | Required for all state-based numeric sensors. | |

### 3.13.8 Sensor Hysteresis Query (SHQ)

| | | |
|---|---|---|
| **Description:** | Query the current value of a hardware hysteresis. | |
| **Argument(s):** | <none> | |
| **Return Value:** | Hysteresis Value (Integer) | The current hysteresis value. The interpretation of this value depends on the sensor object type. |
| **Notes:** | Required for state-based numeric sensors that support hardware hysteresis. This value applies to all defined thresholds. | |

### 3.13.9 Sensor Hysteresis Control (SHC)

| | | |
|---|---|---|
| **Description:** | Modify the hardware hysteresis value. | |
| **Argument(s):** | Hysteresis Value (Integer) | The new hysteresis value. The interpretation of this value depends on the sensor object type. |
| **Return Value:** | <none> | |
| **Notes:** | Required for state-based numeric sensors that support hardware hysteresis. This value applies to all defined thresholds. | |

## 3.14 Data Structures

### 3.14.1 SENSOR_INFO

This data structure specifies the general characteristics of a sensor object. It consists of several *header* fields and a variable number of additional property fields. The variable nature of the property fields is intended to allow flexibility in describing the characteristics of sensor hardware while maintaining prudent use of BIOS memory resources.

| Offset | Name | Length | Value | Description |
|---|---|---|---|---|
| 0x00 | Object Type | BYTE | 0x01 | This field specifies the type of manageability object. Use the *Value* field from Table 1. The value 0x01 to indicate that this is a sensor object. |
| 0x01 | Version | BYTE | 0x10 | This field specifies the version of the Metolious Specification that this manageability object interface is compliant with. The major version is specified in the high nibble, the minor version in the low nibble. For example, the value 0x10 identifies the interface defined in version 1.0 of this specification. |
| 0x02 | Major Sensor Type | BYTE | ENUM | This field specifies the major sensor object type. Use the *Major Type Value* field from Table 4. For example, 0x04 would be used for a physical security sensor. |
| 0x03 | Minor Sensor Type | BYTE | ENUM | This field specifies the minor sensor object type. Use the *Minor Type Value* field from Table 4. For example, 0x01 would be used for a chassis intrusion sensor. |
| 0x04 | Sensor Instance | BYTE | Varies | This field specifies a unique instance for this sensor. Note that this value is zero based (e.g. 0x00 is the first instance) and values are incremented within the context of a major/minor sensor type pair. |
| 0x05 | Hardware Capability | BYTE | Bit Field | This field specifies the basic sensor hardware capabilities. See section 3.14.1.1. |
| 0x06 | Monitored Device Type | BYTE | ENUM | This field specifies the type of device being monitored by this sensor. See section 3.14.1.2. |
| 0x07 | Monitored Device Instance | BYTE | Varies | This field specifies the instance of the device being monitored by this sensor. Note that this value is zero based (e.g. 0x00 is the first instance) and values are incremented within the context of a monitored device type. |
| 0x08 | Property Count | BYTE | Varies | The number (n) of SENSOR_PROPERTY elements existing in the property array. |
| 0x09 + (n-1) * 5 | Property Array | Varies | Varies | An array of 5-byte SENSOR_PROPERTY elements describing additional characteristics of this sensor object. See section 3.14.2. |

The following is a 'C'-style definition of the SENSOR_INFO structure.

```
#define ANYSIZE_ARRAY 1

struct SENSOR_INFO
{
    BYTE ObjectType;
```

```
    BYTE Version;
    BYTE MajorSensorType;
    BYTE MinorSensorType;
    BYTE SensorInstance;
    BYTE HardwareCapability;
    BYTE MonitoredDeviceType;
    BYTE MonitoredDeviceInstance;
    BYTE PropertyCount;
    SENSOR_PROPERTY PropertyArray[ANYSIZE_ARRAY];
};
```

See section 3.14.2 for details concerning the SENSOR_PROPERTY structure. The use of the ANYSIZE_ARRAY is simply for 'C' syntactical correctness. It is the responsibility of upper-level software to parse the sensor property array to extract the required information. Consumers of this array should not assume any ordering of the elements, but can assume that only one instance of a given sensor property type will exist in the array.

#### *3.14.1.1 Hardware Capability*

This 8-bit value specifies the hardware capabilities of this sensor. Possible values for this bit field are defined below. A set bit (1) indicates that a sensor supports the associated hardware capability, while a cleared bit (0) indicates that the capability is not supported.

| Bits | Name | Description |
|---|---|---|
| Bit 0 | Numeric | Numeric sensor. Setting this bit implies that the SR( ) control method and numeric sensor properties are supported. |
| Bit 1 | State-Based | State-based sensor. Setting this bit implies that the SS( ) control method and state-based sensor properties are supported.<br>Setting both bits 0 and 1 indicates a state-based numeric sensor. |
| Bit 2 | Thresholds | Hardware threshold support. Setting this bit implies that the STQ( ), STC( ), and threshold-specific state-based sensor properties are supported. Only valid for state-based numeric sensors (cleared otherwise). |
| Bit 3 | Hysteresis | Hardware hysteresis support. Setting this bit implies that the SHQ( ) and SHC( ) control methods are supported. Only valid for state-based numeric sensors (cleared otherwise). |
| Bit 4 | Local Alerts | The sensor generates a local hardware alert following any state transitions. Only valid for state-based and state-based numeric sensors. |
| Bit 5 | Remote Alerts | The sensor generates a remote hardware alert following any state transitions. Only valid for state-based and state-based numeric sensors. |
| Bit 6 | Manual Re-Arm | The sensor requires manual re-arming following a state transition. Setting this bit implies the SRA( ) control method is supported. |
| Bit 7 | <Reserved> | Cleared (0). |

For example, a chassis intrusion sensor (state-based non-numeric) that is able to generate remote alerts and requires a manual re-arm would have a hardware capability value of 0x52.

#### *3.14.1.2 Monitored Device Type*

This 8-bit value specifies the type of device being monitored by this sensor. Possible values for this enumeration are defined below.

| Value | Name | Description |
|---|---|---|
| | ***Physical Containers*** | |
| 0x00 | System Chassis | System chassis. |
| 0x01 | Docking Station | Docking station. |
| 0x02 | Port Replicator | Port replicator. |

| | | |
|---|---|---|
| 0x03 | Peripheral Bay | Peripheral expansion bay. |
| 0x04 – 0x0F | <Reserved> | Reserved for future standard physical container types. |
| ***System Boards*** | | |
| 0x10 | System Board | System board. |
| 0x11 | Motherboard | Motherboard. |
| 0x12 | Management Module | System management module. |
| 0x13 – 0x1F | <Reserved> | Reserved for future standard system board types. |
| ***Processor/Packaging*** | | |
| 0x20 | Processor | Processor. |
| 0x21 | Processor Module | Processor module. |
| 0x22 | Processor Cartridge | Processor cartridge. |
| 0x23 | Processor Board | Processor board. |
| 0x24 – 0x2F | <Reserved> | Reserved for future standard processor/packaging types. |
| ***Power Sources*** | | |
| 0x30 | Power Source | Power source. |
| 0x31 | Battery | Battery power source. |
| 0x32 | Battery Charger | Battery charger. |
| 0x33 – 0x3F | <Reserved> | Reserved for future standard power source types. |
| ***Memory Devices*** | | |
| 0x40 | Memory Device | Physical memory device. |
| 0x41 – 0x4F | <Reserved> | Reserved for future standard memory device types. |
| ***Mass Storage*** | | |
| 0x50 | Disk Drive | Disk drive. |
| 0x51 – 0x5F | <Reserved> | Reserved for future standard mass storage types. |
| ***Communication Devices*** | | |
| 0x60 | Communication Device | Communication device. |
| 0x61 | Network Adapter | Network adapter. |
| 0x62 – 0x6F | <Reserved> | Reserved for future standard communication device types. |
| ***Cooling Devices*** | | |
| 0x70 | Cooling Device | Cooling device. |
| 0x71 | Fan | Fan cooling device. |
| 0x72 – 0x7F | <Reserved> | Reserved for future standard cooling device types. |
| ***Unknown*** | | |
| 0x80 | Unknown | Unknown device type. |
| 0x81 – 0x8F | <Reserved> | Reserved. |
| ***Vendor-Defined*** | | |
| 0xF0 – 0xFF | <Vendor-defined> | Vendor-defined device types. |

## 3.14.2 SENSOR_PROPERTY

The SENSOR_PROPERTY structure is used to specify additional properties of a sensor object, and provides a foundation for building vendor-defined properties. As stated previously, an array of SENSOR_PROPERTY elements exists within the SENSOR_INFO structure.

The following is a 'C'-style definition of the SENSOR_PROPERTY structure.

```
struct SENSOR_PROPERTY
{
   BYTE   PropertyID;               // Property Type ID
   BYTE   PropertyValue[4];   // Value of the Property
}
```

Note: To avoid issues related to endian-ness for PropertyValue[4] refer to section 1.3.2.2.

### *3.14.2.1 Property Types*

Details on the standard sensor property types are provided below. Properties are grouped according to the sensor classification they are associated with. Note that values not listed in this table are reserved for future standard property types.

| ***Properties for Numeric Sensors*** | | | |
|---|---|---|---|
| **ID** | **Name** | **Status** | **Description** |
| 0x00 | <Reserved> | | |
| 0x01 | Nominal Reading | REQUIRED | The normal value for the sensor reading. Must be specified for numeric sensors. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x02 | Maximum Reading | Optional | The physical maximum value readable by this sensor. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x03 | Minimum Reading | Optional | The physical minimum value readable by this sensor. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x04 | Accuracy | Optional | The accuracy for readings from this sensor. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x05 | Resolution | Optional | The resolution for readings from this sensor. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x06 | Tolerance | Optional | The tolerance for readings from this sensor. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x07 | Non-Linear? | Optional | 0x00000001 (TRUE) if readings from this sensor are not linear throughout the sensor's range. 0x00000000 (FALSE) otherwise. The absence of this property implies a FALSE value. |
| 0x08 – 0x0F | <Reserved> | Reserved for future numeric sensor properties. | |

| ***Properties for State-Based Sensors*** | | | |
|---|---|---|---|
| **ID** | **Name** | **Status** | **Description** |
| 0x10 | State Capability | REQUIRED | A bitmap that specifies the state capability of this sensor. Must be specified for state-based sensors. See section 3.14.2.1.1. |
| 0x11 | Polling Frequency | Optional | The recommended frequency in seconds. This value is used to encourage upper level software to use a value appropriate to this sensor's hardware capabilities (and the overall platform design) when implementing a poll-based event generation policy. |
| 0x12 – 0x1F | <Reserved> | Reserved for future state-based sensor properties. | |

| *Properties for State-Based Numeric Sensors* | | | |
|---|---|---|---|
| **ID** | **Name** | **Status** | **Description** |
| 0x20 | Threshold Capability | REQUIRED | A bitmap that specifies the threshold capability of this sensor. Must be specified if this sensor supports thresholds. See section 3.14.2.1.2. |
| 0x21 | Default Upper Warning Threshold | Optional | The default value for the upper warning (non-critical) threshold. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x22 | Default Lower Warning Threshold | Optional | The default value for the lower warning (non-critical) threshold. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x23 | Default Upper Critical Threshold | Optional | The default value for the upper critical threshold. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x24 | Default Lower Critical Threshold | Optional | The default value for the lower critical threshold. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x25 | Default Upper Fatal Threshold | Optional | The default value for the upper fatal threshold. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x26 | Default Lower Fatal Threshold | Optional | The default value for the lower fatal threshold. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x27 | Default Hysteresis | Optional | The default value for sensor hysteresis. The interpretation of this value depends on the sensor type (see section 3.11). |
| 0x27 – 0x2F | <Reserved> | Reserved for future state-based numeric sensor properties. | |

| *General Sensor Properties* | | | |
|---|---|---|---|
| **ID** | **Name** | **Status** | **Description** |
| 0x30 | SMBIOS Table Entry Reference | Optional | The unique handle of an SMBIOS structure representing the device being monitored by this sensor. For example, a thermal sensor may reference the processor it monitors by referencing a Processor Information (Type 4) structure in the system's SMBIOS table. See section 3.10.5. |
| 0x31 – 0x3F | <Reserved> | Reserved for future general sensor properties. | |

| *Vendor-defined Sensor Properties* | | | |
|---|---|---|---|
| **ID** | **Name** | **Status** | **Description** |
| 0x80+ | <Vendor-defined> | | |

#### *3.14.2.1.1 State Capability*

This 32-bit value specifies the states supported by a sensor object (see Table 2). This property is required for state-based sensors. A set bit (1) indicates that a sensor supports the associated state, while a cleared bit (0) indicates that the state is not supported. Note that state-based sensors must support at least two states.

| **Bits** | **Name** | **Description** |
|---|---|---|
| Bit 0 | OK | The normal operating state. |
| Bit 1 | Warning | Warning (non-critical) state. |
| Bit 2 | Critical | Critical state. |
| Bit 3 | Fatal | Fatal (non-recoverable) state. |
| Bits 4:15 | <Reserved> | Cleared (0). |
| Bits 16:30 | <Vendor-defined> | Vendor-defined sensor states. Cleared (0) when not defined. |
| Bit 31 | <Reserved> | Cleared (0). |

For example, a standard chassis intrusion sensor (described in section 3.11.4.1) that only supports the *OK* and *Warning* states would have a state capability value of 0x00000003.

#### *3.14.2.1.2 Threshold Capability*

This 32-bit value specifies the hardware threshold capabilities for state-based numeric sensors. This property is required for state-based numeric sensors. A set bit (1) indicates that a sensor supports the associated threshold, while a cleared bit (0) indicates that the threshold is not supported.

| Bits | Name | Description |
|---|---|---|
| Bit 0 | Upper Warning | Upper warning (non-critical) threshold. |
| Bit 1 | Lower Warning | Lower warning (non-critical) threshold. |
| Bit 2 | Upper Critical | Upper critical threshold. |
| Bit 3 | Lower Critical | Lower critical threshold. |
| Bit 4 | Upper Fatal | Upper fatal (non-recoverable) threshold. |
| Bit 5 | Lower Fatal | Lower fatal (non-recoverable) threshold. |
| Bits 6:15 | <Reserved> | Cleared (0). |
| Bits 16:31 | <Vendor-defined> | Vendor-defined thresholds. Cleared (0) when not defined. |

For example, the National Semiconductor LM75 ASIC includes a single thermal sensor that supports one upper threshold ($T_{OS}$). Assuming $T_{OS}$ will be used as the upper critical threshold, the threshold capability value for this device would be 0x00000004.

# 3.15 Watchdog Objects

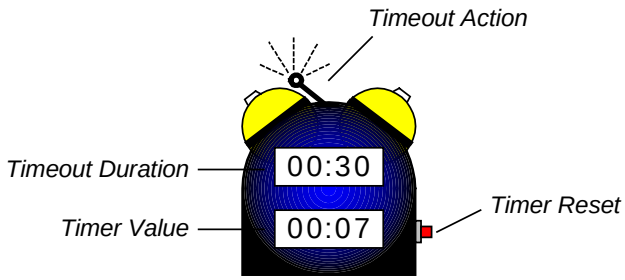
When a serious problem such as a system hang occurs, the ability to detect, analyze, and recover from it can provide a crucial advantage towards the overall manageability of a platform. The hardware used to facilitate these kinds of system health capabilities is often referred to as a *watchdog*. Through the definition of *watchdog objects* in ACPI, this specification facilitates the coordinated use of watchdog hardware between ACPI and manageability, and presents an interface for making watchdog information and control universally available to management software.

This chapter provides details on how to abstract standard watchdog hardware and presents a foundation for building vendor-defined watchdog types.

# 3.16 Watchdog Objects - Overview

A watchdog is defined as any hardware-based device that monitors various elements of system health through the use of a timer. Watchdogs can be thought of as an alarm clock that, unless it is reset every so often, results in a system health-related action when a timeout occurs. The type of action to be taken depends on the type of watchdog. A diagram of the basic abstraction provided by this specification for watchdog hardware is shown in Figure 8.

**Figure 8:** Illustration of a Watchdog Object




## 3.16.1 Hardware Alerting

Watchdogs supporting hardware alerts relieve higher-level consumers from implementing poll-based event generation policies, and generally facilitate a much more responsive and accurate environment. Watchdogs generate hardware alerts upon timer expiration. For example, an OS hang watchdog supporting remote alerting would generate an alert whenever a timeout occurs. Remote consumers would interpret this alert as a signal that the local OS has stopped responding.

As described in section 3.6, this specification defines local hardware alerting as the ability for manageability hardware to generate an ACPI-visible asynchronous notification (e.g. SCI) which then can be decoded to a specific watchdog and then routed up the manageability software stack. Remote alerting is defined as the ability for manageability hardware to directly send an alert to a remote consumer over a LAN, alphanumeric paging, or other data communication medium.

# 3.17 Type Details

This specification defines several classes of watchdog types, representing the most common manageability watchdog hardware available for PC platforms, as shown below in Table 5. OEMs with platforms using hardware watchdogs that are not covered in this specification are encouraged to use vendor-defined types. Details on each watchdog type are provided in the remainder of this section.

**Table 5:** Standard Watchdog Types

| Major Type Value | Hardware ID | Major Watchdog Type | Minor Type Value | Minor Watchdog Type | ID |
|---|---|---|---|---|---|
| 0x00 | MGMT200 | <Reserved> | | | |
| 0x01 | MGMT201 | System Hang Watchdogs | 0x00 | <Reserved> | – |
| | | | 0x01 | System Initialization Failure | I |
| | | | 0x02 | Pre-OS Boot Failure | B |
| | | | 0x03 | OS Boot Failure | L |
| | | | 0x04 | OS Hang | O |
| | | | 0x05 | Shutdown Failure | S |
| 0x02 | MGMT202 | Heartbeat Watchdogs | 0x00 | <Reserved> | – |
| | | | 0x01 | LAN Presence Heartbeat Watchdog | H |
| 0x03 – 0x7F | MGMT203 – MGMT27F | <Reserved for Future Standard Watchdogs> | | | |
| 0x80+ | MGMT280 – MGMT2FF | <Vendor-Defined Watchdogs> | | | |

## 3.17.1 System Hang Watchdogs

System hang watchdogs are defined as timer-based hardware elements that are used to detect and respond to a system hang. The timeout value for system hang watchdogs specifies the amount of time (in seconds) a given process has before a hang is assumed to have occurred. Note that it may be prudent to use the same hardware to implement all of the supported system hang watchdog types on a given platform. In this case the hardware would change roles in response to a system state change (e.g. logically transition from an OS boot failure watchdog to an OS hang watchdog).

### *3.17.1.1 Action Codes*

The following 32-bit value defines the possible action codes for all system hang watchdogs. A set bit (1) indicates that the associated timeout action is supported, while a cleared bit (0) indicates that the action is not supported.

| Bits | Action Name | Description |
|---|---|---|
| Bit 0 | Remote Alert | Generate a remote hardware alert. |
| Bit 1 | Log Event | Record an event in the local system event log. |
| Bit 2 | Reset | Reset the system. |
| Bit 3 | Power Off | Power off the system. |
| Bit 4 | Power Cycle | Power off the system then power it back on. |
| Bits 5:15 | <Reserved> | Reserved for future standard action codes. Should be cleared (0). |
| Bits 16:30 | <Vendor-defined> | Vendor-defined action codes. Cleared (0) when not defined. |
| Bit 31 | <Reserved> | Reserved. Should be cleared (0). |

### *3.17.1.2 Types*

#### *3.17.1.2.1 System Initialization Failure Watchdog*

System initialization failure watchdogs detect and respond to system hangs that may occur during initialization of the core system components (e.g. CPU, cache, etc.). This watchdog should be enabled at system power-on (or reset) and is active until the pre-OS boot process is initiated. System firmware must reset the timer or disable the watchdog prior to a timeout occurring.

#### *3.17.1.2.2 Pre-OS Boot Failure Watchdog*

Pre-OS boot failure watchdogs detect and respond to system hangs that may occur during the initial boot process. This watchdog should be enabled following system initialization and is active until OS load is initiated. System firmware must reset the timer or disable the watchdog prior to a timeout occurring.

#### *3.17.1.2.3 OS Boot Failure Watchdog*

OS boot failure watchdogs detect and respond to system hangs that may occur during the OS load process. This watchdog should be enabled following pre-OS boot and is active until the OS is operational. OS software is typically responsible for resetting or disabling the watchdog prior to a timeout occurring.

#### *3.17.1.2.4 OS Hang Watchdog*

OS hang watchdogs detect and respond to system hangs that may occur whenever the OS is expected to be operational. This watchdog should be enabled following OS load and is active until OS shutdown is initiated. OS software is typically responsible for resetting or disabling the watchdog prior to a timeout occurring.

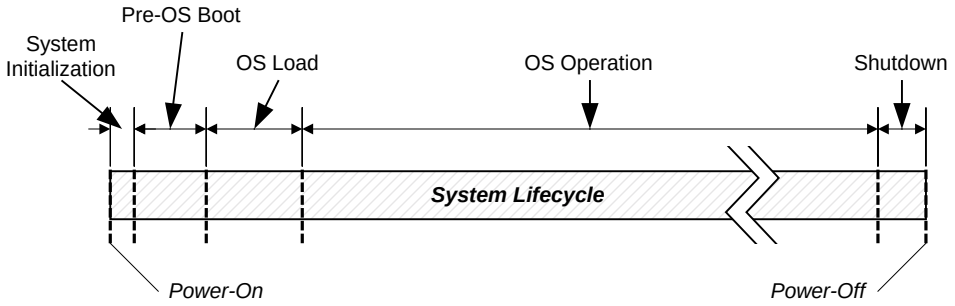
#### *3.17.1.2.5 Shutdown Failure Watchdog*

Shutdown failure watchdogs detect and respond to system hangs that may occur during the system shutdown process. This watchdog should be enabled following the initiation of OS shutdown is active until the system is powered off (or reset). System firmware is typically responsible for resetting or disabling the watchdog prior to a timeout occurring.

### *3.17.1.3 Example*

Figure 9 illustrates the life cycle of an example system that makes use of the five system hang watchdog types described in section 3.17.1.2. Note that a single hardware element is used in this example, where the hardware's role changes as the system proceeds through its lifecycle.

**Figure 9:** System Hang Watchdog Example




At power-on the watchdog hardware initializes and defaults to a timeout value appropriate for the system initialization process. A timeout during this period would indicate a problem with a core system component (e.g. CPU missing), and would trigger the sending of a remote alert followed by a power cycle.

After system initialization the firmware resets the timeout to a value appropriate for the pre-OS boot process, transitioning the watchdog to its new role. A timeout during this period would indicate a problem with the initialization of hardware required to load the OS (e.g. hard disk failure), and would trigger the creation of a new entry in the system event log, the sending of a remote alert, and a power cycle.

Likewise, system firmware and/or OS software would manage the transitions to the OS load, OS operation, and system shutdown roles by setting the hardware's timeout to a value appropriate for the given phase. Timeouts in these phases would be treated similarly to those that occur during the pre-OS boot process.

## 3.17.2 Heartbeat Watchdogs

Heartbeat watchdogs are defined as timer-based hardware elements that are used to notify a consumer of the continued presence and/or operation of a device or system. Note that for this type of watchdog the expiration of the

timer is expected – as this event causes the heartbeat indication to be sent. The timeout value for heartbeat watchdogs specifies the frequency (in seconds) that heartbeat indications should be sent.

#### *3.17.2.1 LAN Presence Heartbeat Watchdog*

LAN presence heartbeat watchdogs proactively notify a consumer that a node is active by sending a 'pong' packet over the network at regular intervals.

The following 32-bit value defines the possible action codes for LAN presence heartbeat watchdogs. A set bit (1) indicates that the associated timeout action is supported, while a cleared bit (0) indicates that the action is not supported.

| Bits | Action Name | Description |
|---|---|---|
| Bit 0 | Heartbeat | Generate presence notification (heartbeat). |
| Bits 1:15 | <Reserved> | Reserved for future standard action codes. Should be cleared (0). |
| Bits 16:30 | <Vendor-defined> | Vendor-defined action codes. Cleared (0) when not defined. |
| Bit 31 | <Reserved> | Reserved. Should be cleared (0). |

# 3.18 ASL Definition

The ACPI definition for watchdog objects is provided below. Details on control method are provided in section 3.19. Details on data structures are provided in section 3.20.

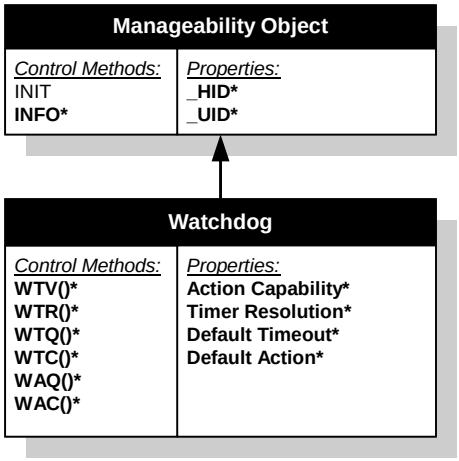
```
Device(<type>W<id>){
   Name(_HID, <hid>)          // Watchdog Hardware ID (PnP ID)
   Name(_UID, <uid>)          // Watchdog Signature
   Method(INIT, 0) {…}        // Watchdog Hardware Initialization
   Method(INFO, 0) {…}        // Watchdog Information
   Method(WTV, 0) {…}         // Watchdog Timer Value
   Method(WTR, 0) {…}         // Watchdog Timer Reset
   Method(WTQ, 0) {…}         // Watchdog Timeout Query
   Method(WTC, 1) {…}         // Watchdog Timeout Control
   Method(WAQ, 0) {…}         // Watchdog Action Query
   Method(WAC, 1) {…}         // Watchdog Action Control
}
```

## 3.18.1 Requirements

Figure 10 expresses the watchdog classifications and associated ASL requirements using an object inheritance paradigm. Derived watchdog 'classes' inherit the control method and property requirements of all parent 'classes'. This diagram is simply intended to illustrate the ASL requirements for watchdog objects, and does not infer an object-oriented development process.

Required control methods and properties for each object type are indicated with an asterisk ('*').

**Figure 10:** Watchdog Requirements Hierarchy




## 3.18.2 Name

Manageability object names, as with any object in ACPI, must follow the ASL naming convention defined in section 15.1.2 of the ACPI Specification. Below is the *recommended* format for specifying the name of a watchdog object. Although any ACPI-compliant name can be used, this format allows watchdogs to be easily identified when inspecting ASL.

**`Device(<type>W<id>)`**

`<type>` A single character used to specify the watchdog object type. This specification recommends using the *ID* field from Table 5. For example the letter `'O'` would be used for an OS hang watchdog.

`W` A single character that specifies the type of manageability object. This specification recommends using the *ID* field from Table 1. The letter `'W'` is used for watchdog objects.

`<id>` A two-character identifier that, together with the previous two characters, creates a device name that is unique throughout all ACPI device objects. This specification recommends using the same value as used for the *Watchdog Instance* field of the `WATCHDOG_INFO` structure (see section 3.20.1).

## 3.18.3 Device Identification

Manageability objects utilize the _HID and _UID device identification objects to facilitate Plug and Play enumeration, providing the capability for the 'automatic' enumeration of each object class by the OS.

**`Name(_HID, "MGMT2XX")`**

Used to specify the Plug and Play hardware ID for this device. See the *Hardware ID* field from Table 5. For example, an OS hang watchdog would have a HID value of "MGMT201".

**`Name(_UID, <uid>)`**

`<uid>` A 32-bit value used to specify a unique identifier for this watchdog object. This specification recommends using a combination of the *Object Type*, *Major Watchdog Type*, *Minor Watchdog Type*, and *Watchdog Instance* fields of the `WATCHDOG_INFO` structure (see section 3.20.1). For example, the first instance of an OS hang watchdog would have a UID value of 0x02010400.

# 3.19 Control Methods

## 3.19.1 Initialize (INIT)

| | |
|---|---|
| **Description:** | This control method is called by a supporting driver immediately after the OS enumerates a watchdog object, and is used to perform any required initialization of the watchdog hardware. |
| **Argument(s):** | <none> |
| **Return Value:** | <none> |
| **Notes:** | Optional. Implement only if hardware must be initialized prior to its use. |

## 3.19.2 Information (INFO)

| | | |
|---|---|---|
| **Description:** | Return a `WATCHDOG_INFO` structure describing the general properties of a watchdog object. | |
| **Argument(s):** | <none> | |
| **Return Value:** | Watchdog Information (Buffer) | A `WATCHDOG_INFO` structure as a byte array. See section 3.20.1. |
| **Notes:** | Required for all watchdogs. | |

## 3.19.3 Watchdog Timer Value (WTV)

| | | |
|---|---|---|
| **Description:** | Returns the current value of the watchdog timer. | |
| **Argument(s):** | <none> | |
| **Return Value:** | Timer Value (Integer) | The current timer value (in seconds). Note that the value 0x80000000 indicates that the timeout duration is unknown or currently unavailable. |
| **Notes:** | Required for all watchdogs. | |

## 3.19.4 Watchdog Timer Reset (WTR)

| | |
|---|---|
| **Description:** | Reset the value of the watchdog timer to the current timeout duration. |
| **Argument(s):** | <none> |
| **Return Value:** | <none> |
| **Notes:** | Required for all watchdogs. The watchdog client should call this control method before the timer expires to prevent the watchdog from implementing the timeout action. |

## 3.19.5 Watchdog Timeout Query (WTQ)

| | | |
|---|---|---|
| **Description:** | Query the current watchdog timeout duration. | |
| **Argument(s):** | <none> | |
| **Return Value:** | Timeout Duration (Integer) | The current timeout duration (in seconds). This value must be formulated as some multiple of the watchdog resolution. See the *Timer Resolution* property in section 3.20.2.1.<br>Note that the value 0x80000000 indicates that the timeout duration is unknown or currently unavailable. |
| **Notes:** | Required for all watchdogs. | |

### 3.19.6 Watchdog Timeout Control (WTC)

| | | |
|---|---|---|
| **Description:** | Modify the watchdog timeout duration. | |
| **Argument(s):** | Timeout Duration (Integer) | The new timeout duration (in seconds). This value must be formulated as some multiple of the watchdog resolution. See the *Timer Resolution* property in section 3.20.2.1. |
| **Return Value:** | <none> | |
| **Notes:** | Required for all watchdogs. | |

### 3.19.7 Watchdog Timeout Action Query (WAQ)

| | | |
|---|---|---|
| **Description:** | Query the action to be taken when a watchdog timeout occurs. | |
| **Argument(s):** | <none> | |
| **Return Value:** | Action Code (Integer) | The current timeout action code. The interpretation of this value depends on the watchdog type (see section 3.17).<br>Note that the value 0x00000000 (no action) indicates that the watchdog is disabled. The value 0x80000000 indicates that the timeout action is unknown or currently unavailable. |
| **Notes:** | Required for all watchdogs. | |

### 3.19.8 Watchdog Timeout Action Control (WAC)

| | | |
|---|---|---|
| **Description:** | Modify the action to be taken when a watchdog timeout occurs. | |
| **Argument(s):** | Action Code (Integer) | The new action code, specifying the operation to be taken upon the expiration of the watchdog timer. The interpretation of this value depends on the watchdog type (see section 3.17).<br>Note that the value 0x00000000 (no action) disables a watchdog. |
| **Return Value:** | <none> | |
| **Notes:** | Required for all watchdogs. | |

## 3.20 Data Structures

### 3.20.1 WATCHDOG_INFO

This data structure specifies the general characteristics of a watchdog object. It consists of several *header* fields and a variable number of additional property fields. The variable nature of the property fields is intended to allow flexibility in describing the characteristics of watchdog hardware while maintaining prudent use of BIOS memory resources.

| Offset | Name | Length | Value | Description |
|---|---|---|---|---|
| 0x00 | Object Type | BYTE | 0x02 | This field specifies the type of manageability object. Use the *Value* field from Table 1. The value 0x02 to indicate that this is a watchdog object. |
| 0x01 | Version | BYTE | 0x10 | This field specifies the version of the Metolious Specification that this manageability object interface is compliant with. The major version is specified in the high nibble, the minor version in the low nibble. For example, the value 0x10 identifies the interface defined in version 1.0 of this specification. |
| 0x02 | Major Watchdog Type | BYTE | ENUM | This field specifies the major watchdog object type. Use the *Major Type Value* field from Table 5. For example, 0x01 would be used for a system hang watchdog. |
| 0x03 | Minor | BYTE | ENUM | This field specifies the minor watchdog object type. Use the *Minor* |

| | | | | |
|---|---|---|---|---|
| | Watchdog Type | | | *Type Value* field from Table 5. For example, 0x02 would be used for an OS hang watchdog. |
| 0x04 | Watchdog Instance | BYTE | Varies | This field specifies a unique instance for this watchdog. Note that this value is zero based (e.g. 0x00 is the first instance) and values are incremented within the context of a major/minor watchdog type pair. |
| 0x05 | Property Count | BYTE | Varies | The number (n) of WATCHDOG_PROPERTY elements existing in the property array. |
| 0x06 + (n-1) * 5 | Property Array | Varies | Varies | An array of 5-byte WATCHDOG_PROPERTY elements describing additional characteristics of this watchdog object. See section 3.20.2. |

The following is a 'C'-style definition of the WATCHDOG_INFO structure.

```
#define ANYSIZE_ARRAY 1

struct WATCHDOG_PROPERTY
{
    BYTE ObjectType;
    BYTE Version;
    BYTE MajorWatchdogType;
    BYTE MinorWatchdogType;
    BYTE WatchdogInstance;
    BYTE PropertyCount;
    WATCHDOG_PROPERTY PropertyArray[ANYSIZE_ARRAY];
};
```

See section 3.20.2 for details concerning the WATCHDOG_PROPERTY structure. The use of the ANYSIZE_ARRAY is simply for 'C' syntactical correctness. It is the responsibility of upper-level software to parse the watchdog property array to extract the required information. Consumers of this array should not assume any ordering of the elements, but can assume that only one instance of a given watchdog property type will exist in the array.

## 3.20.2 WATCHDOG_PROPERTY

The WATCHDOG_PROPERTY structure is used to specify additional properties of a watchdog object, and provides a foundation for building vendor-defined properties. As stated previously, an array of WATCHDOG_PROPERTY elements exists within the WATCHDOG_INFO structure.

The following is a 'C'-style definition of the WATCHDOG_PROPERTY structure.

```
struct WATCHDOG_PROPERTY
{
    BYTE   PropertyID;                    // Property Type ID
    BYTE   PropertyValue[4];          // Value of the Property
}
```

Note: To avoid issues related to endian-ness for PropertyValue[4] refer to section 1.3.2.2.

### *3.20.2.1 Property Types*

Details on the standard watchdog property types are provided below. Properties are grouped according to the watchdog classification they are associated with.

| ***General Watchdog Properties*** | | | |
|---|---|---|---|
| **Value** | **Name** | **Status** | **Description** |
| 0x00 | <Reserved> | | |
| 0x01 | Action Capability | REQUIRED | This 32-bit value specifies the timeout action(s) supported by a watchdog object. This value is based on the action codes presented for each watchdog type in section 3.17.<br>For example, a standard OS hang watchdog that was capable of resetting the system (bit 0) or powering off the system (bit 1) would have an action |

| | | | |
|---|---|---|---|
| | | | capability value of 0x00000003. |
| 0x02 | Timer Resolution | REQUIRED | The resolution (in seconds) for watchdog timer and timeout duration values. |
| 0x03 | Default Timeout | REQUIRED | The default watchdog timeout value (in seconds). |
| 0x04 | Default Action | REQUIRED | A bitmap specifying the default action(s) to take upon the expiration of the watchdog timer. This value is based on the action codes presented for each watchdog type in section 3.17. |
| 0x05 – 0x0F | <Reserved> | Reserved for future standard watchdog properties. | |

| ***Vendor-Defined Watchdog Properties*** | | | |
|---|---|---|---|
| **ID** | **Name** | **Status** | **Description** |
| 0x80+ | <Vendor-defined> | | |

# 4 Platform Management – Extended Interface

## 4.1 Overview

The Metolious extended interface provides a uniform host interface to manageability hardware over the wide range of volume server platforms. This interface collects the server management hardware into a set of abstracted interfaces and exposes these interfaces as manageability objects in ACPI. By abstracting the details of server management hardware, this specification isolates the management software from details of the hardware and allows the software that works across the range of server platforms. The standardization of the management interface to host software also enables a single driver model that works regardless of the capabilities of the system interface device. Exposing the Metolious extended interfaces via ACPI further allows plug and play enumeration of management hardware and enables the management software to automatically configure to the dynamic management hardware environment.

## 4.2 Platform Management Hardware

Platform management hardware consists of entities that monitor, control and report the health of the system. These include devices such as sensors, watchdog timers, and other system control devices. In addition, system management information is also maintained in sensor event logs, FRU EEPROM, and sensor information storage.

### 4.2.1 Sensors

Sensors are the staple of platform management hardware. Sensors monitor the health of the system and report any excursions from the normal. The most common sensors include temperature sensors, voltage sensors, fan sensors, and chassis intrusion sensors. Other sensors such as DIMM presence sensor, processor presence sensor, power supply redundancy sensor are usually specific to server platforms.

Sensors are classified according to the type of reading they provide and/or the type of events they generate. A sensor can return either an analog or discrete reading. Sensor events could be discrete or threshold-based.

Sensors have associated properties depending on the type of the sensor or their event generation capability. Properties of analog sensors include tolerance, resolution, rate unit, linearity, and maximum/minimum reading.

The properties associated with a sensor cannot be deduced from the sensor hardware alone and hence must be supplied separately.

### 4.2.2 Management Information Store

Management Information consists of persistent data such as system event log, FRU information, and Sensor related information.

System Event Log consists of critical events that are captured for later analysis. Critical events represent drastic changes in the monitored environment. Processor over-temperature, fan failure, chassis intrusion detection are some examples of critical events. Such events are typically logged to a non-volatile store.

Server class systems typically have FRU information associated with the major system components. The FRU information includes asset tags, and identification information such as serial numbers, part number, and type. FRU information is maintained in a non-volatile storage such as an EEPROM.

Sensors information includes the properties of sensors such as threshold levels, accuracy, …Further sensors have to be correlated to the monitored hardware and the FRU entities associated with such hardware. Such information needs to be maintained in the system and delivered to system software to enable effective use of a sensor.

### 4.2.3 System Recovery & Control Devices

Recovery devices such as watchdogs monitor, detect, correct system failure and return the system to an operational state. Watchdog uses a timer to monitor the system health. Expiry of a watchdog timer could be associated with a number of recovery actions such as reset, power off, power cycle, generate alert.

### 4.2.4 Dynamic Management Hardware

Enterprise-class server platforms are modular systems that allow addition/removal of components. The management hardware population on such systems could change dynamically. For example, additional sensors may show up as a result of adding/replacing a RAID backplane. Also, with the advent of hot pluggable devices, new sensors could dynamically appear/disappear behind such devices.

## 4.3 Architectural Goals for Extended Interface

The Metolious extended interface definition seeks to satisfy the following goals:

1. Provide standardized, abstracted, message-based interface to enumerate, access, and control platform management hardware on high availability server platforms
2. Provide support for IPMI-based management hardware on server platforms
3. Provide a straightforward implementation for IPMI-based server management hardware
4. Provide support for dynamic management hardware population
5. Provide a low level messaging interface to support OEM specific management hardware and management interfaces

## 4.4 Problem Statement

Platform management hardware such as sensors and watchdog timers could be directly exposed to the host software. High availability server systems, however, may have a large number of systems due to the size and complexity of the hardware. Server specific sensors such as DIMM presence sensor, Power Supply redundancy status sensor, and processor presence sensor also contribute to the increased sensor population. Hence, exposing the sensor hardware directly would add complexity to the management interface.

Further, Server management hardware may contain dynamic management hardware population. This requires a management interface abstraction that can handle such management hardware.

## 4.5 Solution

The Metolious extended interface abstracts the server management hardware into a set of well-defined interfaces and exposes these interfaces as ACPI objects. The implementation of the interface hides the details and complexity of the management hardware behind it.

The extended interface comprises of the following component interfaces:

- ***Sensor Interface*** – abstracts access to sensor hardware. This interface associates each sensor with a unique sensor number. All sensors present can be accessed and controlled by the interface commands using the sensor number as a parameter.
- ***Sensor Information Interface*** – abstracts information related to sensors such as sensor type, characteristics, and association of sensor to the managed entities and FRU information. This abstracted sensor information is presented by this interface as a set of numbered sensor data records (SDR). The interface allows for enumerating the sensor data records (SDR) and for accessing a specific SDR.
- ***System Event Log Interface*** – abstracts the event-logging interface. System Event Log (SEL) captures critical events for "post mortem" analysis to determine the cause of the failure. This interface abstracts access to the SEL via a set of interface commands that allow system software to read or delete individual SEL entries as well as clears the entire log.
- ***FRU Interface*** – abstracts access to FRU device that contains system component information such as asset tags, model number. Multiple FRU devices are supported with each FRU device uniquely identified by a FRU device number.
- ***Recovery & Control Interface*** – abstracts access to watchdog timers and similar platform alerting h/w. This interface allows setting of the watchdog timeout interval and the action associated with the timeout.

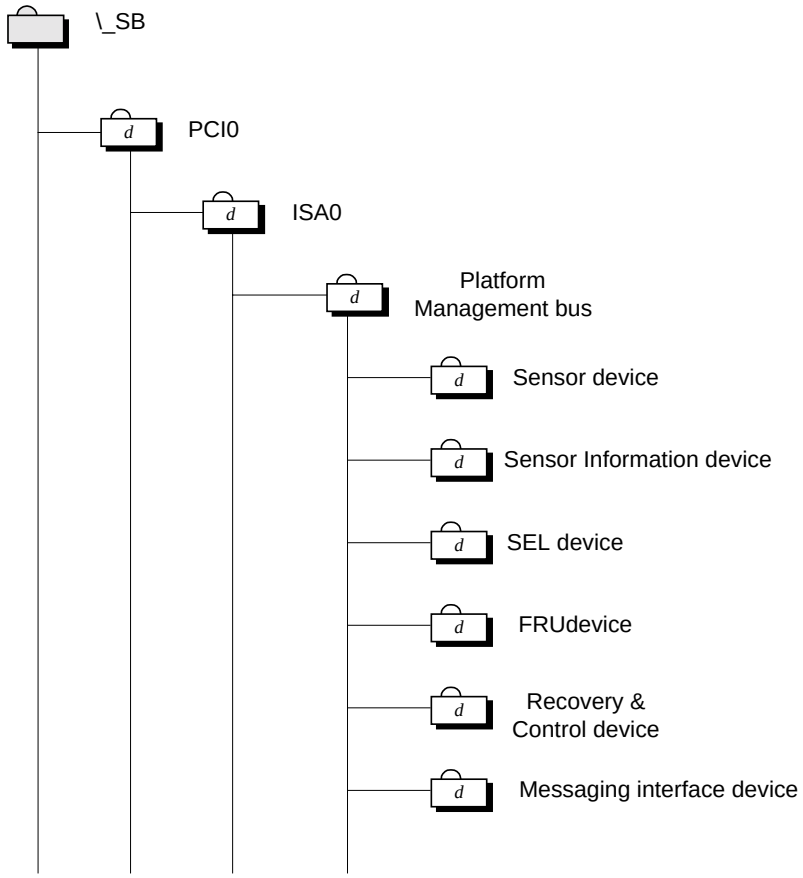
- ***Messaging Interface*** – supports a low level messaging to platform management subsystem. This interface is defined to support vendor specific commands and command formats. The data fields of the interface functions are implementation specific.

While the above interfaces could be represented in a proprietary fashion to the system software, exposing the interfaces using ACPI provides a flexible, extendible and OS-neutral solution based on a proven configuration interface.

The namespace of ACPI is a bus-device hierarchy that represents the functional hierarchy of the system. The namespace begins with the system bus and shows each device or bus under its parent bus.

In order to fit under the ACPI model, the Metolious extended interface is modeled as manageability device objects. The collection of these manageability devices could be viewed as a bus, namely Platform Management bus. The following figure shows the representation of Metolious extended interface device objects in the ACPI name space.

**Figure 11:** ACPI Name Space Hierarchy for Extended Interface



# 5 Manageability objects for Extended Interface

ACPI Manageability objects defined corresponds to the management interfaces defined in the previous section. Each manageability object is associated with a plug and play identifier and provides a standard interface to system software via a set of control methods.

This section describes the control methods associated with each manageability object and the associated data structures.

## 5.1 Device Identification

These device identifiers enable 'plug and play' enumeration of the objects by the OS. The plug and play identifier associated with each extended interface manageability objects is given in Table 6. ACPI object _HID is used to associate the plug and play identifier with the device object.

**Table 6: Manageability Object Types - Extended Interface**

| Value | Hardware ID | Manageability object type | Description |
|---|---|---|---|
| 0x30 | MGMT300 | Sensor Interface object | Abstracts sensor hardware |
| 0x31 | MGMT310 | Sensor Information Interface object | Abstracts information related to sensors such as calibration parameters |
| 0x32 | MGMT320 | SEL Interface object | Abstracts event logging interface |
| 0x33 | MGMT330 | FRU Interface object | Abstracts interface to asset tags and system related information of FRUs |
| 0x34 | MGMT340 | Recovery & Control object | Abstracts interface to watchdog timers and system control h/w |
| 0x35 | MGMT350 | Messaging interface object | Low level messaging interface that supports unformatted 'raw' interface to management hardware |

## 5.2 ACPI Specifics

Metolious management interface is defined on top of ACPI specification, and therefore limited by the capabilities of ACPI Interface. Some of the ACPI conventions and restrictions that affect the Metolious definition are given below:

- All ACPI names are restricted to 32-bits (4 characters)
- ACPI Control Methods can handle up to seven parameters. Each argument is an object, and could in turn be a "package[1]" style object that refers to other objects. Hence, packages are used whenever the number of arguments is large.
- An ACPI Control method is allowed to return exactly one object upon completion. If more than one return parameter is needed, a package of objects is returned.

## 5.3 Commands & Completion Codes

All control methods defined in this specification (other than INIT and _STA) either implicitly or explicitly specify a command to the platform management hardware. Explicit commands are specified as part of the arguments to the control methods. Control methods (except INIT, _STA) also return a completion code. This code is returned as the first parameter of the return value package. Commands that complete normally return 0x00 to indicate normal completion. All non-zero completion codes correspond to errors. The completion codes for an IPMI based

[1] A package is an unnamed collection of data items, constants, and/or references to control methods

implementation is fully specified by the completion codes of the IPMI specification. Some of the key completion codes are listed below:

**Table 7: Completion Codes**

| Code | Definition |
|---|---|
| 0x00 | Command Completed Normally |
| 0xC1 | Invalid Command |
| 0xC5 | Reservation Canceled or invalid reservation ID |
| 0xFF | Unspecified Error |

# 5.4 Sensor Interface Device

The ACPI definition for the sensor interface device is given below.

```
Device (SENS) {
      Name(_HID, 0)           // PnP ID
      Method(INIT, 0) {…}     // one time device specific initialization
      Method(_STA, 0) {…}     //device status: interface enabled/disabled
      Method(SHQ, 2) {…}      // Get Sensor Hysteresis
      Method(SHC, 4) {…}      // Set Sensor Hysteresis
      Method(STQ, 1) {…}      // Get  Sensor Threshold
      Method(STC, 8) {…}      // Set Sensor Threshold
      Method(SR, 1) {…}       // Get Sensor Reading
      Method(SRA, 6) {…}      // Re-arm Sensor Events
      Method(SEEQ, 1) {…}     // Get Sensor Event Enable
      Method(SEEC, 2) {…}     // Set Sensor Event Enable
}
```

## 5.4.1 Control Methods for Sensor Interface Device

**INIT**

| | |
|---|---|
| **Description** | **Called by the device driver to perform one-time initialization of the device** |
| **Argument(s)** | None |
| **Return Value** | None |
| **Notes** | Optional for IPMI-based implementations. Mandatory for other implementations |

**_STA**

| | |
|---|---|
| **Description** | **Returns status for the sensor interface device** |
| **Argument(s)** | None |
| **Return Value** | Bit 0: Set if device is present<br>Bit 1: Set if device is enabled<br>Bit 2: Reserved<br>Bit 3: Set if device is functioning properly<br>Bit 4-31: Reserved |
| **Notes** | Mandatory command |

**SHQ**

| | | |
|---|---|---|
| **Description** | **Query the current value of hardware hysteresis for specified sensor** | |
| **Argument(s)** | Sensor Number | Between 0 and 0xFF. Sensor Number obtained from reading the Sensor Information device |
| | Sensor Mask | For future definition, set to 0xFF |
| **Return Value** | Completion code | |
| | Positive-going threshold hysteresis | |
| | Negative-going threshold hysteresis | |
| **Notes** | | |

**SHC**

| | | |
|---|---|---|
| **Description** | **Modify the sensor hysteresis value for specified sensor** | |
| **Argument(s)** | Sensor Number | Between 0 and 0xFF. Sensor Number obtained from reading the Sensor Information device |
| | Sensor Mask | For future definition, set to 0xFF |
| | Positive-going threshold hysteresis value | Set to 0x00, if positive-going threshold is not supported |
| | Negative-going threshold hysteresis value | Set to 0x00, if positive-going threshold is not supported |
| **Return Value** | Completion code | |
| **Notes** | | |

**STQ**

| | | |
|---|---|---|
| **Description** | **Query the current Threshold value for specified sensor** | |
| **Argument(s)** | Sensor Number | Between 0 and 0xFF. Sensor Number obtained from reading the Sensor Information device |
| **Return Value** | Completion code | |
| | Bit 7:6 Reserved set to 0x00<br>Bit 5 – upper non-recoverable threshold<br>Bit 4 – upper critical threshold<br>Bit 3 – upper non-critical threshold<br>Bit 2 – lower non-recoverable threshold<br>Bit 1 – lower critical threshold<br>Bit 0 – lower non-critical threshold | This bit mask indicates readable thresholds |
| | Lower non-critical threshold | Ignore if the corresponding bit mask is not set |
| | Lower critical threshold | |
| | Lower non-recoverable threshold | |
| | Upper non-critical threshold | |
| | Upper critical threshold | |
| | Upper non-recoverable threshold | |
| **Notes** | | |

**STC**

| **Description** | **Modify the current Threshold value for specified sensor** | |
|---|---|---|
| **Argument(s)** | Sensor Number | Between 0 and 0xFF. Sensor Number obtained from reading the Sensor Information device |
| | Bit 7:6 Reserved set to 0x00<br>Bit 5 – upper non-recoverable threshold<br>Bit 4 – upper critical threshold<br>Bit 3 – upper non-critical threshold<br>Bit 2 – lower non-recoverable threshold<br>Bit 1 – lower critical threshold<br>Bit 0 – lower non-critical threshold | This bit mask indicates readable thresholds |
| | Lower non-critical threshold | Ignore if the corresponding bit mask is not set |
| | Lower critical threshold | |
| | Lower non-recoverable threshold | |
| | Upper non-critical threshold | |
| | Upper critical threshold | |
| | Upper non-recoverable threshold | |
| **Return Value** | Completion code | |
| **Notes** | | |

**SR**

| **Description** | **Returns the current reading for the specified sensor** | |
|---|---|---|
| **Argument(s)** | Sensor Number | Between 0 and 0xFF. Sensor Number obtained from reading the Sensor Information device |
| **Return Value** | Completion code | |
| | Sensor reading (ignore this field if the sensor is not an analog sensor | |
| | Bit 7: 0= Event messages disabled from this sensor<br>Bit 6: 0=Sensor scanning is disabled<br>Bit 5: 1=Initial update in progress, a sensor “re-arm” command has been issued to request update of sensor status and the update is still pending<br>Bit 4:0 Reserved | |
| | For Threshold-based sensors<br>Bit 7:6 reserved<br>Bit 5: at or above (>=) upper non-recoverable threshold<br>Bit 4: at or above (>=) upper critical threshold<br>Bit 3: at or above (>=) upper non-critical threshold<br>Bit 2: at or above (>=) lower non-recoverable threshold<br>Bit 1: at or above (>=) lower critical threshold<br>Bit 0: at or above (>=) lower non critical threshold<br><br>For discrete reading sensors<br>Bit 7: state 7 asserted<br>…<br>Bit 0: state 0 asserted | |
| | For discrete reading sensors only<br>Bit 7: Reserved<br>Bit 6: state 14 asserted<br>…<br>Bit 0: state 8 asserted | |
| **Notes** | | |

**SRA**

| | |
|---|---|
| **Description** | **Re-arm sensor** |
| **Argument(s)** | Sensor Number |
| | Bit 7: 0=re-arm all event status from this sensor<br>Bit 6:0 reserved |
| | For Threshold-based sensors<br>Bit 7:6 reserved<br>Bit 5: re-arm upper non-recoverable threshold assertion event<br>Bit 4: re-arm upper critical threshold assertion event<br>Bit 3: re-arm upper non-critical threshold assertion event<br>Bit 2: re-arm lower non-recoverable threshold assertion event<br>Bit 1: re-arm lower critical threshold assertion event<br>Bit 0: re-arm lower non critical threshold assertion event<br><br>For discrete reading sensors<br>Bit 7: re-arm assertion event for state bit 7<br>…<br>Bit 0: re-arm assertion event for state bit 0 |
| | For discrete reading sensors<br>Bit 7: Reserved<br>Bit 6: re-arm assertion event for state bit 14<br>…<br>Bit 0: re-arm assertion event for state bit 8 |
| | For Threshold-based sensors<br>Bit 7:6 reserved<br>Bit 5: re-arm upper non-recoverable threshold deassertion event<br>Bit 4: re-arm upper critical threshold deassertion event<br>Bit 3: re-arm upper non-critical threshold deassertion event<br>Bit 2: re-arm lower non-recoverable threshold deassertion event<br>Bit 1: re-arm lower critical threshold deassertion event<br>Bit 0: re-arm lower non critical threshold deassertion event<br><br>For discrete reading sensors<br>Bit 7: re-arm deassertion event for state bit 7<br>…<br>Bit 0: re-arm deassertion event for state bit 0 |
| | For discrete reading sensors<br>Bit 7: Reserved<br>Bit 6: re-arm deassertion event for state bit 14<br>…<br>Bit 0: re-arm deassertion event for state bit 8 |
| **Return Value** | Completion code |
| **Notes** | |

### 5.4.1 Data Structures for Sensor Interface Device

The data structures for the sensor interface device are the data structures from IPMI for the corresponding function

## 5.5 Sensor Information Interface Device

This device is referred to as Sensor Data Record or SDR in the definitions below. The ACPI definition for the sensor interface device is given below.

```
Device (SINF) {
```

```
        Name(_HID, 0)           // PnP ID
        Method(INIT, 0) {…}     // one time device specific initialization
        Method(_STA, 0) {…} //device status: interface enabled/disabled
        Method(SDRQ, 1) {…}     // Get SDR
        Method(SDRR, 0) {…}     // Reserve SDR
        Method(SDRS, 0) {…}     // Get SDR Status (SDR Repository Info)
}
```

### 5.5.1 Control Methods for Sensor Interface Device

**INIT**

| | |
|---|---|
| **Description** | **Called by the device driver to perform one-time initialization of the device** |
| **Argument(s)** | None |
| **Return Value** | None |
| **Notes** | Optional for IPMI-based implementations. Mandatory for other implementations |

**_STA**

| | |
|---|---|
| **Description** | **Returns status for the sensor interface device** |
| **Argument(s)** | None |
| **Return Value** | Bit 0: Set if device is present<br>Bit 1: Set if device is enabled<br>Bit 2: Reserved<br>Bit 3: Set if device is functioning properly<br>Bit 4-31: Reserved |
| **Notes** | Mandatory command |

**SDRQ**

| | |
|---|---|
| **Description** | **Queries the SDR for a specified record** |
| **Argument(s)** | Reservation ID, LS Byte (if not a partial read, use 0x0000 for reservation ID) |
| | Reservation ID, MS Byte |
| | Record ID for record to get, LS Byte |
| | Record ID for record to get, MS Byte |
| | Offset into record |
| | Bytes to read (0xFF indicates read entire record) |
| **Return Value** | Completion code |
| | Record ID for next record, LS Byte |
| | Record ID for next record, MS Byte |
| | 1:N bytes Record Data |
| **Notes** | Record ID of 0x0000 returns first record, and 0x0FFFF returns the last record |

**SDRR**

| | |
|---|---|
| **Description** | **Reserve SDR Repository** |
| **Argument(s)** | None |
| **Return Value** | Completion code |
| | Reservation ID, LS Byte |
| | Reservation ID, MS Byte |
| **Notes** | Mandatory command. Enables partial reads of SDR |

**SDRS**

| | | |
|---|---|---|
| **Description** | **Get SDR Status** | |
| **Argument(s)** | None | |
| **Return Value** | Completion Code | |
| | Record count – LS Byte | Number of records in the repository |
| | Record count – MS Byte | |
| | Most recent addition time stamp | Time stamp is an unsigned 32-bit value of local time as number seconds from 00:00:00, January 1, 1970 |
| | Most recent erase time stamp | |
| **Notes** | Mandatory command. Allows system s/w to find out if the sensor population has changed. s/w required to re-enumerate if the timestamp returned different from cached timestamp | |

#### 5.5.1.1 Reservation Command

Reservation mechanism is used to notify the caller of the Metolious extended interface of a concurrent access to the device that may have invalidated the enumerated record identifiers. This command protects the clients who perform multi-part read of the sensor information record from reading incorrect information. If the sensor information record has changed during a multi-part read, the next partial read command would be rejected with the indication that the reservation identifier of the caller is no longer valid. The caller infers that the records has changed, rejects the data read so far, and restarts the multi-part read after acquiring a new reservation.

### 5.5.2 Data Structures for Sensor Information Device

The data structures for the sensor interface device are defined under the section titled ‘Sensor Data Record Formats’ in the IPMI v1.0 Specification.

## 5.6 SEL Interface Device

The ACPI definition for SEL interface device is given below.

```
Device (SEL) {
      Name(_HID, 0)           // PnP ID
      Method(INIT, 0) {…}     // one time device specific initialization
      Method(_STA, 0) {…}     //device status: interface enabled/disabled
      Method(SELI, 2) {…}     // Get SEL Info
      Method(SELQ, 2) {…}     // GET SEL ENTRY
      Method(SELA, 1) {…}     // SET SEL INFO–Add,
      Method(SELD, 1) {…}     // SET SEL INFO–Delete,
      Method(SELC, 1) {…}     // SET SEL INFO–Clear,
      Method(SELR, 2) {…}     // Reserve SEL
}
```

### 5.6.1 Control Methods for SEL Interface Device

**INIT**

| | |
|---|---|
| **Description** | **Called by the device driver to perform one-time initialization of the device** |
| **Argument(s)** | None |
| **Return Value** | None |
| **Notes** | Optional for IPMI-based implementations. Mandatory for other implementations |

**_STA**

| | |
|---|---|
| **Description** | **Returns status for the SEL interface device** |
| **Argument(s)** | None |
| **Return Value** | Bit 0: Set if device is present<br>Bit 1: Set if device is enabled<br>Bit 2: Reserved<br>Bit 3: Set if device is functioning properly<br>Bit 4-31: Reserved |
| **Notes** | Mandatory command |

**SELI**

| | |
|---|---|
| **Description** | **Returns information regarding the SEL Interface** |
| **Argument(s)** | None |
| **Return Value** | Completion Code |
| | SEL Version |
| | Number of log entries in SEL – LS Byte |
| | Number of log entries in SEL – MS Byte |
| | Free space in bytes – LS Byte |
| | Free space in bytes – MS Byte |
| | Bytes 7:10 Most recent addition timestamp |
| | Bytes 11:14 Most recent erased timestamp |
| | Byte 15: Operation Support<br>Bit 7: Overflow flag – events dropped due to lack of space in SEL<br>Bit 6:4 reserved<br>Bit 3: Individual SEL entries can be deleted using SELC<br>Bit 2: Partial SEL add supported using SELC and SELR<br>Bit 1: SELR command supported<br>Bit 0: Reserved |
| **Notes** | |

**SELQ**

| **Description** | **Queries for a specific SEL Entry, partial reads are allowed** | |
|---|---|---|
| **Argument(s)** | Reservation ID, LS Byte | Only required for partial reads |
| | Reservation ID, MS Byte | |
| | SEL Record ID, LS Byte | 0x0000 = GET FIRST ENTRY |
| | SEL Record ID, MS Byte | 0xFFFF = GET LAST ENTRY |
| | Offset into record | |
| | Bytes to read | 0xFF means read entire record |
| **Return Value** | Completion Code | |
| | Next SEL Record ID, LS Byte | |
| | Next SEL Record ID, MS Byte | |
| | Record Data, 16 byte buffer | |
| **Notes** | | |

**SELA**

| **Description** | **Add a SEL Entry** | |
|---|---|---|
| **Argument(s)** | Reservation ID, LS Byte | Only required for partial add. Use 0x0000 for Reservation ID, otherwise |
| | Reservation ID, MS Byte | |
| | Record ID, LS Byte | Only required for partial add. Use 0x0000 for Record ID, Otherwise |
| | Record ID, MS Byte | |
| | Offset into record | |
| | In Progress | 0x00 = partial add in progress<br>0x01 = partial add complete |
| | Record Data | 16 byte buffer |
| **Return Value** | Completion Code | |
| | Record ID for added record, LS Byte | |
| | Record ID for added record, MS Byte | |
| **Notes** | | |

**SELD**

| **Description** | **Delete a SEL Entry** | |
|---|---|---|
| **Argument(s)** | Reservation ID, LS Byte | |
| | Reservation ID, MS Byte | |
| | Record ID to delete, LS Byte | 0x0000 = FIRST_ENTRY<br>0xFFFF = LAST_ENTRY |
| | Record ID to delete, MS Byte | |
| **Return Value** | Completion code | |
| | Record ID for deleted record, LS Byte | |
| | Record ID for deleted record, MS Byte | |
| **Notes** | | |

**SELC**

| | |
|---|---|
| **Description** | **Clear SEL Entry** |
| **Argument(s)** | Reservation ID, LS Byte |
| | Reservation ID, MS Byte |
| | Command Code<br>0x00 = Get Erasure status<br>0xAA = Initiate Erase |
| **Return Value** | Completion code |
| | Erasure Status<br>0x00 = erasure in progress<br>0x01 = erase complete |
| **Notes** | |

**SELR**

| | |
|---|---|
| **Description** | **Reserve SEL** |
| **Argument(s)** | None |
| **Return Value** | Completion Code |
| | Reservation ID, LS Byte |
| | Reservation ID, MS Byte |
| **Notes** | Mandatory command for IPMI implementations. Enables partial add of SEL entry, partial reads of SEL entry |

### 5.6.2 Data Structures for SEL Interface Device

The SEL Interface Device abstracts the SEL Information as a list of SEL Records. The format of an individual SEL Record is defined under the section titled 'SEL Record Formats' in the IPMI Specification.

## 5.7 FRU Interface Device

The ACPI definition for FRU interface device is given below.

```
Device (FRU) {
      Name(_HID, 0)           // PnP ID
      Method(INIT, 0) {…}     // one time device specific initialization
      Method(_STA, 0) {…}     //device status: interface enabled/disabled
      Method(FRUI, 2) {…}     // Get FRU Inventory Area Info
      Method(FRUQ, 2) {…}     // Read FRU Inventory Data
      Method(FRUC, 1) {…}     // Write FRU Inventory Data
}
```

### 5.7.1 Control Methods for FRU Interface Device

**INIT**

| | |
|---|---|
| **Description** | **Called by the device driver to perform one-time initialization of the device** |
| **Argument(s)** | None |
| **Return Value** | None |
| **Notes** | Optional for IPMI-based implementations. Mandatory for other implementations |

**_STA**

| | |
|---|---|
| **Description** | **Returns status for the sensor interface device** |
| **Argument(s)** | None |
| **Return Value** | Bit 0: Set if device is present<br>Bit 1: Set if device is enabled<br>Bit 2: Reserved<br>Bit 3: Set if device is functioning properly<br>Bit 4-31: Reserved |
| **Notes** | Mandatory command |

**FRUI**

| | | |
|---|---|---|
| **Description** | **Returns FRU Inventory Area Information** | |
| **Argument(s)** | None | |
| **Return Value** | FRU Inventory area size in bytes, LS Byte | |
| | FRU Inventory area size in bytes, MS Byte | |
| | Access Type | Bit 7:1 reserved<br>Bit 0: 0 = byte access, 1 = word access |
| **Notes** | | |

**FRUQ**

| | |
|---|---|
| **Description** | **Read FRU Inventory for specified FRU Device at specified offset** |
| **Argument(s)** | FRU Device ID |
| | FRU Inventory offset to read, LS Byte |
| | FRU Inventory offset to read, MS Byte |
| | Count to read |
| **Return Value** | Completion code |
| | Count returned |
| | FRU Data (returned as a buffer) |
| **Notes** | |

**FRUC**

| | |
|---|---|
| **Description** | **Write FRU Inventory for specified FRU device at specified offset** |
| **Argument(s)** | FRU Device ID |
| | FRU Inventory Offset to write, LS Byte |
| | FRU Inventory Offset to write, MS Byte |
| | FRU Data (represented as a buffer) |
| **Return Value** | Completion code |
| | Count written |
| **Notes** | |

## 5.7.2 Data Structures for FRU Interface Device

The FRU Interface Device provides control methods to read/write FRU Inventory data. The record format of the FRU Inventory data exposed by the FRUQ, FRUC control methods is specified in the Platform Management FRU Information Storage Definition.

## 5.8 Recovery & Control Interface Device

The ACPI definition for the Recovery & Control interface device is given below.

```
Device (RC) {
      Name(_HID, 0)           // PnP ID
      Method(INIT, 0) {…}     // one time device specific initialization
      Method(_STA, 0) {…}     //device status: interface enabled/disabled
      Method(WDQ, 2) {…}      // Get Watchdog Timer
      Method(WDC, 2) {…}      // Set/Reset Watchdog Timer
}
```

### 5.8.1 Control Methods for Recovery & Control Interface Device

**INIT**

| | |
|---|---|
| **Description** | **Called by the device driver to perform one-time initialization of the device** |
| **Argument(s)** | None |
| **Return Value** | None |
| **Notes** | Optional for IPMI-based implementations. Mandatory for other implementations |

**_STA**

| | |
|---|---|
| **Description** | **Returns status for the sensor interface device** |
| **Argument(s)** | None |
| **Return Value** | Bit 0: Set if device is present<br>Bit 1: Set if device is enabled<br>Bit 2: Reserved<br>Bit 3: Set if device is functioning properly<br>Bit 4-31: Reserved |
| **Notes** | Mandatory command |

**WDQ**

| | |
|---|---|
| **Description** | **Queries the Watchdog Timer** |
| **Argument(s)** | None |
| **Return Value** | Completion Code |
| | Timer Use<br>Bit 7:3 – reserved<br>Bit 2:0<br>000b = reserved<br>001b = BIOS FRB2<br>010b = BIOS/POST<br>011b = OS Load<br>100b = SMS/OS<br>101b = OEM |
| | Timer Actions<br>Bit 7: reserved<br>Bit 6:4 pre-timeout interrupt<br>000b = none<br>001b = SMI<br>010b = NMI<br>Bit 3 : reserved<br>Bit 2:0 timeout action<br>000b = no action<br>001b = Hard reset<br>010b = Power Down<br>011b = Power cycle |
| | Pre-timeout interval in seconds |
| | Timer use expiration flags<br>('1' indicates timer expired while associated 'use' was selected)<br>Bit 7:6 reserved<br>Bit 5: OEM<br>Bit 4: SMS/OS<br>Bit 3: OS load<br>Bit 2: BIOS/POST<br>Bit 1: BIOS FRB2 |
| | Initial countdown value, LS Byte (100ms/count) |
| | Initial countdown value, MS Byte |
| | Present countdown value, LS Byte |
| | Present countdown value, MS Byte |
| **Notes** | |

**WDC**

| **Description** | **Set/Reset the Watchdog Timer** |
|---|---|
| **Argument(s)** | Command code<br>0 = Set Watchdog timer<br>1 = Reset Watchdog timer<br>(for reset command, the remaining argument fields are 0) |
| | Timer Use<br>Bit 7:3 – reserved<br>Bit 2:0<br>000b = reserved<br>001b = BIOS FRB2<br>010b = BIOS/POST<br>011b = OS Load<br>100b = SMS/OS<br>101b = OEM |
| | Timer Actions<br>Bit 7: reserved<br>Bit 6:4 pre-timeout interrupt<br>000b = none<br>001b = SMI<br>010b = NMI<br>Bit 3 : reserved<br>Bit 2:0 timeout action<br>000b = no action<br>001b = Hard reset<br>010b = Power Down<br>011b = Power cycle |
| | Pre-timeout interval in seconds |
| | Timer use expiration flags<br>1 = clear time use expiration bit<br>bit 7:6 reserved<br>bit 5: OEM<br>bit 4: SMS/OS<br>bit 3: OS load<br>bit 2: BIOS/POST<br>bit 1: BIOS FRB2 |
| | Initial countdown value, LS Byte (100ms/count) |
| | Initial countdown value, MS Byte |
| **Return Value** | Completion code |
| **Notes** | |

### 5.8.2 Data Structures for Recovery & Control Interface Device

Refer to section titled 'BMC Watchdog Timer Commands' in the IPMI v1.0 Specification, for additional details regarding the arguments and return values of the control methods WDQ, and WDC.

## 5.9 Messaging Interface Device

The ACPI definition for the Messaging interface device is given below.

```
Device (MSG) {
      Name(_HID, 0)           // PnP ID
      Method(INIT, 0) {…}     // one time device specific initialization
      Method(_STA, 0) {…}     //device status: interface enabled/disabled
      Method(MSGS, 2) {…}     // Send Message
      Method(MSGG, 2) {…}     // Get Message
```

}

### 5.9.1 Control Methods for Messaging Interface Device

**INIT**

| | |
|---|---|
| **Description** | **Called by the device driver to perform one-time initialization of the device** |
| **Argument(s)** | None |
| **Return Value** | None |
| **Notes** | Optional for IPMI-based implementations. Mandatory for other implementations |

**_STA**

| | |
|---|---|
| **Description** | **Returns status for the sensor interface device** |
| **Argument(s)** | None |
| **Return Value** | Bit 0: Set if device is present<br>Bit 1: Set if device is enabled<br>Bit 2: Reserved<br>Bit 3: Set if device is functioning properly<br>Bit 4-31: Reserved |
| **Notes** | Mandatory command |

**MSGS**

| | |
|---|---|
| **Description** | **Send proprietary message to Platform Management Controller** |
| **Argument(s)** | Size of buffer |
| | Message data (format of data bytes is proprietary to the implementation) |
| **Return Value** | Completion Code |
| **Notes** | |

**MSGG**

| | |
|---|---|
| **Description** | **Get proprietary message from Platform Management Controller** |
| **Argument(s)** | |
| **Return Value** | Completion Code |
| | Size of buffer |
| | Message data (format of returned message bytes is proprietary to the implementation) |
| **Notes** | MSG devices initiates this command in response to indication from the embedded controller that there is a message pending for this device |

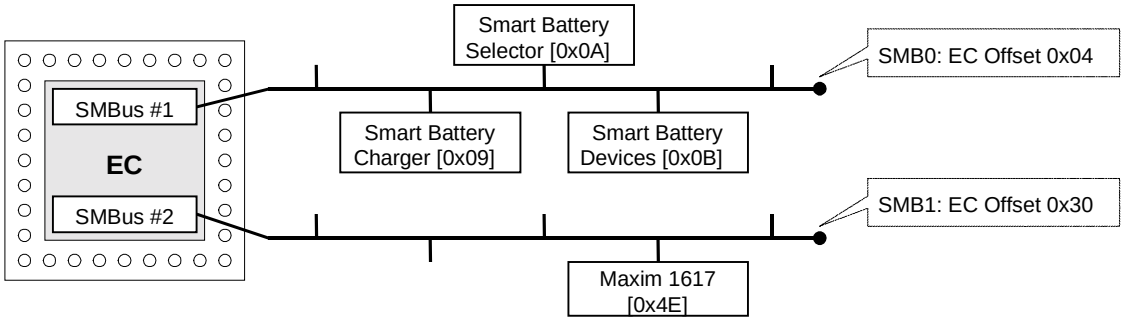
# Appendix A Sample ASL

## A.1 Mobile Example

### A.1.1 Overview

The sample ASL presented in this section was tested on an Intel Mobile Reference Platform. This system includes a Maxim 1617 ASIC (providing two thermal sensors) which is connected to the system using an ACPI-compliant embedded controller (EC) SMBus. Figure 12 illustrates the platform manageability hardware configuration for this system.

**Figure 12:** Mobile Example System




Information on the embedded controller is available in chapter 13 of the ACPI specification:
http://www.teleport.com/~acpi/

Information on the SMBus is available at:
http://www.sbs-forum.org/smbus/specs/

Information on the Maxim 1617 is available at:
http://www.maxim-ic.com/arpdf/1855.pdf

### EC SMBus

Implementing EC SMBus access from AML is required to communicate with the Maxim 1617 ASIC and successfully model the sensors it contains. As shown in Figure 12 the reference platform includes two SMBus interfaces within the EC. The Smart Battery Subsystem is connected to the SMBus at EC offset 0x04, while the Maxim 1617 is connected to the SMBus at EC offset 0x30.

As discussed in section 3.3, access to the SMBus must be coordinated between the various SMBus consumers. To facilitate this, the interface defined in the *SMBus Control Method Interface Specification* has been employed in this sample ASL to provide synchronization between AML and OS software. Developers should obtain the latest version of the SMBus CMI specification and model their implementation accordingly.

A full listing of the ASL to implement this functionality is provided below.

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//                                EC-SMBus                                //
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Device:      SMB0
// ------------------------------------------------------------------------
// Description: The first EC-SMBus segment (includes the SBS).
////////////////////////////////////////////////////////////////////////////
Device(SMB0)
{
    Name(_HID, "SMBUS01")
    Name(_UID, 0)

    Mutex(SBX0, 0)          // SMBus transactional synchronization.

    //
    // OperationRegion:
    // ----------------
    // This SMBus resides at offset 0x04 in EC space.  See the ACPI
    // Specification for information on the EC-SMBus register interface.
    //
    OperationRegion(SMB0,EmbeddedControl,0x04,0x40)
    Field(SMB0,ByteAcc,Lock,Preserve)
    {
        PRTC,   8,     // Protocol
        STS,    8,     // Status
        ADDR,   8,     // Address
        CMD,    8,     // Command
        DATA,   256,   // SMBus Data Bytes (Block)
        BCNT,   8,     // Block Count
        AADR,   8,     // Alarm Address
        ADB0,   8,     // Alarm Data Byte 0
        ADB1,   8      // Alarm Data Byte 1
    }
    Field(SMB0,ByteAcc,Lock,Preserve)
    {
        Offset(4),  // Move to byte offset 4 (beginning of data).
        DAT0, 8,    // 8-bit data register for byte/word access.
        DAT1, 8     // 8-bit data register for word access.
    }

    //
    // _SBI (SMBus Information):
    // -------------------------
    // Returns a SMB_INFO structure describing the properties of this
    // SMBus segment.
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = SMBus Control Method Interface (CMI) version (Integer)
    //  (1)     = SMB_INFO data structure (Buffer)
    //
    Method(_SBI)
    {
        //
        // Local0 is the return package.  The CMI version is set to v1.0 (0x10)
        //
        Store(Package(2){0x10,0x00}, Local0)

        Store(Buffer()
        {
            0x10,                   // SMB_INFO structure Version (v1.0)
            0x10,                   // SMBus Specification Version (v1.0)
            0x00,                   // Segment Hardware Capability
            0x00,                   // Alert Polling Interval (Supports Async Notifications)
            0x03,                   // Device Count
```

```
        0x09, 0x00,             // SMBus Device #1: SBS Charger
        0x00, 0x00, 0x80, 0x86, //      SMB_UDID... (Vendor ID is a placeholder)
        0x00, 0x01, 0x00, 0x00,
        0x53, 0x42, 0x53, 0x09,
        0x00, 0x00, 0x00, 0x00,
        0x0A, 0x00,             // SMBus Device #2: SBS Selector
        0x00, 0x00, 0x80, 0x86, //      SMB_UDID... (Vendor ID is a placeholder)
        0x00, 0x02, 0x00, 0x00,
        0x53, 0x42, 0x53, 0x0A,
        0x00, 0x00, 0x00, 0x00,
        0x0B, 0x00,             // SMBus Device #3: SBS Battery Devices
        0x00, 0x00, 0x80, 0x86, //      SMB_UDID... (Vendor ID is a placeholder)
        0x00, 0x03, 0x00, 0x00,
        0x53, 0x42, 0x53, 0x0B,
        0x00, 0x00, 0x00, 0x00
    }, Index(Local0, 1))
    Return(Local0)
} // _SBI

//
// SWTC (Wait for Transaction Complete):
// -------------------------------------
// Wait until the previous SMBus transaction has completed.
//
// Parameters:
//  Arg0    = Timeout Value (in ms)
//
// Return Value:
//  0x00    = OK
//  0x07    = Unknown Failure
//  0x10    = Address Not Acknowledged
//  0x11    = Device Error
//  0x12    = Command Access Denied
//  0x13    = Unknown Error
//  0x17    = Device Access Denied
//  0x18    = Timeout
//  0x19    = Unsupported Protocol
//  0x1A    = Bus Busy
//  0x1F    = PEC (CRC-8) Error
//
Method(SWTC, 1)
{
    Store(Arg0, Local0)
    Store(0x07, Local2)

    //
    // The previous command has completed when the protocol
    // register is equal to 0 (zero).  Wait <timeout> ms
    // (in 10ms chunks) for this to occur.
    //
    Store(1, Local1)
    While(LEqual(Local1, 1))
    {
        If(LEqual(PRTC, 0))
        {
            And(STS, 0x1F, Local2)      // Store status code.
            Store(0x00, Local1)         // Terminate loop.
        }
        Else
        {
            //
            // Transaction isn't complete.  Check for timeout, and if not,
            // sleep 10ms and loop again.
            //
            If(LLess(Local0, 10))
            {
                Store(0x18, Local2)     // ERROR: Timeout occurred.
                Store(0x00, Local1)     // Terminate loop.
            }
            Else
            {
```

```
                Sleep(10)
                Subtract(Local0, 10, Local0)
            }
        }
    }

    Return(Local2)
} // Method(SWTC)

//
// _SBR (SMBus Read):
// ------------------
//
// Parameters:
//  Arg0    = Protocol (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command (Integer)
//
// Return Value (Package):
//  (0)     = Status (Integer)
//  (1)     = Data Length (Integer)
//  (2)     = Data (Integer | Buffer)
//
Method(_SBR, 3)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(3){0x07,0x00,0x00}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking.
    //
    If(LNotEqual(Arg0, 0x03))                   // Read Quick
    {
        If(LNotEqual(Arg0, 0x05))               // Receive Byte
        {
            If(LNotEqual(Arg0, 0x07))           // Read Byte
            {
                If(LNotEqual(Arg0, 0x09))       // Read Word
                {
                    If(LNotEqual(Arg0, 0x0B))   // Read Block
                    {
                        Store(0x19, Index(Local0, 0))
                        Return(Local0)
                    }
                }
            }
        }
    }

    //
    // Acquire the SMBus mutex to ensure transactional synchronization.
    //
    If(LEqual(Acquire(SBX0, 0xFFFF), 0))
    {
        //
        // Make sure the SMBus is ready for this transaction.  If not,
        // return a 'bus busy' error code.  Note that 'we' should be
        // the only consumer...
        //
        If(LNotEqual(PRTC, 0))
        {
            Store(0x1A, Index(Local0, 0))   // ERROR: Bus is busy.
        }
        Else
        {
            //
```

```
            // Initiate the transaction by writing the slave address,
            // command, and protocol registers.  Note that the command
            // code is always written, even if the protocol (e.g. 'read
            // quick') doesn't require it (it will be ignored).  Note also
            // that the “Read Block” always returns 32-byte buffer regardless
            // of the actual block length.  This is not a requirement but is
            // implementation specific to this sample ASL.
            //
            Store(ShiftLeft(Arg1, 1), ADDR)
            Store(Arg2, CMD)
            Store(Arg0, PRTC)

            //
            // Wait for completion.  Save the status code, data size,
            // and data into the return package (if required by the
            // protocol).
            //
            Store(SWTC(1000), Index(Local0, 0))

            If(LEqual(Arg0, 0x05))          // Receive Byte
            {
                Store(1, Index(Local0, 1))
                Store(DAT0, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x07))          // Read Byte
            {
                Store(1, Index(Local0, 1))
                Store(DAT0, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x09))          // Read Word
            {
                Store(2, Index(Local0, 1))
                Store(DAT1, Local1)
                ShiftLeft(Local1, 8, Local1)
                Add(Local1, DAT0, Local1)
                Store(Local1, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x0B))          // Read Block
            {
                Store(BCNT, Index(Local0, 1))
                Store(DATA, Index(Local0, 2))
            }
        }

        Release(SBX0)
    }

    Return(Local0)
} // _SBR()

//
// _SBW (SMBus Write):
// -------------------
//
// Parameters:
//  Arg0    = Protocol Value (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command Code (Integer)
//  Arg3    = Data Length (Integer)
//  Arg4    = Data (Integer | Buffer)
//
// Return Value (Package):
//  (0)     = Status (Integer)
//
Method(_SBW, 5)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(1){0x07}, Local0)
```

```
//
// Make sure the protocol is valid, if not return the
// 'invalid protocol' status code.  Note that this segment
// does not support packet error checking or the 'process call'
// protocol.
//
If(LNotEqual(Arg0, 0x02))                   // Write Quick
{
    If(LNotEqual(Arg0, 0x04))               // Send Byte
    {
        If(LNotEqual(Arg0, 0x06))           // Write Byte
        {
            If(LNotEqual(Arg0, 0x08))       // Write Word
            {
                If(LNotEqual(Arg0, 0x0A))   // Write Block
                {
                    Store(0x19, Index(Local0, 0))
                    Return(Local0)
                }
            }
        }
    }
}

//
// Acquire the SMBus mutex to ensure transactional synchronization.
//
If(LEqual(Acquire(SBX0, 0xFFFF), 0))
{
    //
    // Make sure the SMBus is ready for this transaction.  If not,
    // return a 'bus busy' error code.  Note that 'we' should be
    // the only consumer...
    //
    If(LNotEqual(PRTC, 0))
    {
        Store(0x1A, Local0)
    }
    Else
    {
        //
        // Initiate the transaction by writing the slave address,
        // command, and protocol registers. Note that the command
        // code and data length are always written, even if the
        // protocol (e.g. 'write quick') doesn't require it (it
        // will be ignored).
        //
        Store(ShiftLeft(Arg1, 1), ADDR)
        Store(Arg2, CMD)
        Store(Arg3, BCNT)

        If(LEqual(Arg0, 0x06))          // Write Byte
        {
            Store(Arg4, DAT0)
        }
        If(LEqual(Arg0, 0x08))          // Write Word
        {
            And(Arg4, 0x00FF, DAT0)
            ShiftRight(Arg4, 8, DAT1)
        }
        If(LEqual(Arg0, 0x0A))          // Write Block
        {
            Store(Arg4, DATA)
        }

        Store(Arg0, PRTC)

        //
        // Wait for completion.
        //
```

```
                Store(SWTC(1000), Local0)
            }
            Release(SBX0)
        }

        Return(Local0)
    } // _SBW()

    //
    // _SBA (SMBus Alert Information):
    // -------------------------------
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = Status Code (Integer) -> {0x00=Success | 0x01=No Active Alert
    //                                      | 0x07=Unknown Failure}
    //  (1)     = Slave Address (Integer)
    //  (2)     = Data Length (Integer)
    //  (3)     = Data (Integer)
    //
    Method(_SBA, 0)
    {
        //
        // Local0 is the return package.  The status code is defaulted
        // to 'unknown failure' (0x07).
        //
        Store(Package(3){0x07,0x00,0x00}, Local0)

        //
        // Acquire the SMBus mutex to ensure transactional synchronization.
        //
        If(LEqual(Acquire(SBX0, 0xFFFF), 0))
        {
            //
            // Make sure there's an active (non-consumed) alarm by
            // checking bit 6 of the status register.  If not, the
            // return package already indicates that there isn't an
            // active alarm.
            //
            If(And(STS, 0x40))
            {
                Store(0x00, Index(Local0, 0))       // Success

                ShiftRight(AADR, 1, Index(Local0, 1))       // Slave Address

                Store(2, Index(Local0, 2))          // Data Length

                Store(ShiftLeft(ADB1, 8), Local1)   // Data
                Add(Local1, ADB0, Local1)
                Store(Local1, Index(Local0, 3))

                //
                // Clear the alarm by resetting the status register.
                //
                Store(0x00, STS)
            }
            Else
            {
                Store(0x01, Index(Local0, 0))       // Status = No Alert
            }

            Release(SBX0)
        }

        Return(Local0)
    } // _SBA()

} // Device(SMB0)
```

```
////////////////////////////////////////////////////////////////////////////
// Device:      SMB1
// ------------------------------------------------------------------------
// Description: The second EC-SMBus segment (includes the Maxim 1617).
////////////////////////////////////////////////////////////////////////////
Device(SMB1)
{
    Name(_HID, "SMBUS01")
    Name(_UID, 1)

    Mutex(SBX1, 0)          // SMBus transactional synchronization.

    //
    // OperationRegion:
    // ----------------
    // This SMBus resides at offset 0x30 in EC space.  See the ACPI
    // Specification for information on the EC-SMBus register interface.
    //
    OperationRegion(SMB1,EmbeddedControl,0x30,0x40)
    Field(SMB1,ByteAcc,Lock,Preserve)
    {
        PRTC,   8,     // Protocol
        STS,    8,     // Status
        ADDR,   8,     // Address
        CMD,    8,     // Command
        DATA,   256,   // SMBus Data Bytes (Block)
        BCNT,   8,     // Block Count
        AADR,   8,     // Alarm Address
        ADB0,   8,     // Alarm Data Byte 0
        ADB1,   8      // Alarm Data Byte 1
    }
    Field(SMB1,ByteAcc,Lock,Preserve)
    {
        Offset(4),  // Move to byte offset 4 (beginning of data).
        DAT0, 8,    // 8-bit data register for byte/word access.
        DAT1, 8     // 8-bit data register for word access.
    }

    //
    // _SBI (SMBus Information):
    // -------------------------
    // Returns a SMB_INFO structure describing the properties of this
    // SMBus segment.
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = SMBus Control Method Interface (CMI) version (Integer)
    //  (1)     = SMB_INFO data structure (Buffer)
    //
    Method(_SBI)
    {
        //
        // Local0 is the return package.  The CMI version is set to v1.0 (0x10)
        //
        Store(Package(2){0x10,0x00}, Local0)

        Store(Buffer()
        {
            0x10,                   // SMBus_INFO structure Version (v1.0)
            0x10,                   // SMBus Specification Version (v1.0)
            0x00,                   // Hardware Capability
            0x0A,                   // Alert Polling Interval (poll every 10 seconds)
            0x01,                   // Device Count
            0x09, 0x00,             // SMBus Device #1: Maxim 1617
            0x00, 0x00, 0x80, 0x86, //      SMB_UDID... (Vendor ID is a placeholder)
            0x00, 0x01, 0x00, 0x00, //                   (Device ID is a placeholder)
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00
```

```
    }, Index(Local0, 1))
    Return(Local0)
} // _SBI

//
// SWTC (Wait for Transaction Complete):
// -------------------------------------
// Wait until the previous SMBus transaction has completed.
//
// Parameters:
//  Arg0    = Timeout Value (in ms)
//
// Return Value:
//  0x00    = OK
//  0x07    = Unknown Failure
//  0x10    = Address Not Acknowledged
//  0x11    = Device Error
//  0x12    = Command Access Denied
//  0x13    = Unknown Error
//  0x17    = Device Access Denied
//  0x18    = Timeout
//  0x19    = Unsupported Protocol
//  0x1A    = Bus Busy
//  0x1F    = PEC (CRC-8) Error
//
Method(SWTC, 1)
{
    Store(Arg0, Local0)
    Store(0x07, Local2)

    //
    // The previous command has completed when the protocol
    // register is equal to 0 (zero).  Wait <timeout> ms
    // (in 10ms chunks) for this to occur.
    //
    Store(1, Local1)
    While(LEqual(Local1, 1))
    {
        If(LEqual(PRTC, 0))
        {
            And(STS, 0x1F, Local2)      // Store status code.
            Store(0x00, Local1)         // Terminate loop.
        }
        Else
        {
            //
            // Transaction isn't complete.  Check for timeout, and if not,
            // sleep 10ms and loop again.
            //
            If(LLess(Local0, 10))
            {
                Store(0x18, Local2)     // ERROR: Timeout occurred.
                Store(0x00, Local1)     // Terminate loop.
            }
            Else
            {
                Sleep(10)
                Subtract(Local0, 10, Local0)
            }
        }
    }

    Return(Local2)
} // Method(SWTC)

//
// _SBR (SMBus Read):
// ------------------
//
// Parameters:
//  Arg0    = Protocol (Integer)
```

```
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command (Integer)
//
// Return Value (Package):
//  (0)     = Status (Integer)
//  (1)     = Data Length (Integer)
//  (2)     = Data (Integer | Buffer)
//
Method(_SBR, 3)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(3){0x07,0x00,0x00}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking.
    //
    If(LNotEqual(Arg0, 0x03))                   // Read Quick
    {
        If(LNotEqual(Arg0, 0x05))               // Receive Byte
        {
            If(LNotEqual(Arg0, 0x07))           // Read Byte
            {
                If(LNotEqual(Arg0, 0x09))       // Read Word
                {
                    If(LNotEqual(Arg0, 0x0B))   // Read Block
                    {
                        Store(0x19, Index(Local0, 0))
                        Return(Local0)
                    }
                }
            }
        }
    }

    //
    // Acquire the SMBus mutex to ensure transactional synchronization.
    //
    If(LEqual(Acquire(SBX1, 0xFFFF), 0))
    {
        //
        // Make sure the SMBus is ready for this transaction.  If not,
        // return a 'bus busy' error code.  Note that 'we' should be
        // the only consumer...
        //
        If(LNotEqual(PRTC, 0))
        {
            Store(0x1A, Index(Local0, 0))   // ERROR: Bus is busy.
        }
        Else
        {
            //
            // Initiate the transaction by writing the slave address,
            // command, and protocol registers.  Note that the command
            // code is always written, even if the protocol (e.g. 'read
            // quick') doesn't require it (it will be ignored).
            //
            Store(ShiftLeft(Arg1, 1), ADDR)
            Store(Arg2, CMD)
            Store(Arg0, PRTC)

            //
            // Wait for completion.  Save the status code, data size,
            // and data into the return package (if required by the
            // protocol).
            //
            Store(SWTC(1000), Index(Local0, 0))
```

```
            If(LEqual(Arg0, 0x05))          // Receive Byte
            {
                Store(1, Index(Local0, 1))
                Store(DAT0, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x07))          // Read Byte
            {
                Store(1, Index(Local0, 1))
                Store(DAT0, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x09))          // Read Word
            {
                Store(2, Index(Local0, 1))
                Store(DAT1, Local1)
                ShiftLeft(Local1, 8, Local1)
                Add(Local1, DAT0, Local1)
                Store(Local1, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x0B))          // Read Block
            {
                Store(BCNT, Index(Local0, 1))
                Store(DATA, Index(Local0, 2))
            }
        }

        Release(SBX1)
    }

    Return(Local0)
} // _SBR()

//
// _SBW (SMBus Write):
// -------------------
//
// Parameters:
//  Arg0    = Protocol Value (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command Code (Integer)
//  Arg3    = Data Length (Integer)
//  Arg4    = Data (Integer | Buffer)
//
// Return Value (Package):
//  (0)     = Status (Integer)
//
Method(_SBW, 5)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(1){0x07}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking or the 'process call'
    // protocol.
    //
    If(LNotEqual(Arg0, 0x02))                   // Write Quick
    {
        If(LNotEqual(Arg0, 0x04))               // Send Byte
        {
            If(LNotEqual(Arg0, 0x06))           // Write Byte
            {
                If(LNotEqual(Arg0, 0x08))       // Write Word
                {
                    If(LNotEqual(Arg0, 0x0A))   // Write Block
                    {
                        Store(0x19, Index(Local0, 0))
```

```
                        Return(Local0)
                    }
                }
            }
        }

        //
        // Acquire the SMBus mutex to ensure transactional synchronization.
        //
        If(LEqual(Acquire(SBX1, 0xFFFF), 0))
        {
            //
            // Make sure the SMBus is ready for this transaction.  If not,
            // return a 'bus busy' error code.  Note that 'we' should be
            // the only consumer...
            //
            If(LNotEqual(PRTC, 0))
            {
                Store(0x1A, Local0)
            }
            Else
            {
                //
                // Initiate the transaction by writing the slave address,
                // command, and protocol registers. Note that the command
                // code and data length are always written, even if the
                // protocol (e.g. 'write quick') doesn't require it (it
                // will be ignored).
                //
                Store(ShiftLeft(Arg1, 1), ADDR)
                Store(Arg2, CMD)
                Store(Arg3, BCNT)

                If(LEqual(Arg0, 0x06))          // Write Byte
                {
                    Store(Arg4, DAT0)
                }
                If(LEqual(Arg0, 0x08))          // Write Word
                {
                    And(Arg4, 0x00FF, DAT0)
                    ShiftRight(Arg4, 8, DAT1)
                }
                If(LEqual(Arg0, 0x0A))          // Write Block
                {
                    Store(Arg4, DATA)
                }

                Store(Arg0, PRTC)

                //
                // Wait for completion.
                //
                Store(SWTC(1000), Local0)
            }
            Release(SBX1)
        }

        Return(Local0)
    } // _SBW()

    //
    // _SBA (SMBus Alert Information):
    // -------------------------------
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = Status Code (Integer) -> {0x00=Success | 0x01=No Active Alert | 0x07=Unknown
Failure}
```

```
    //  (1)     = Slave Address (Integer)
    //  (2)     = Data Length (Integer)
    //  (3)     = Data (Integer)
    //
    Method(_SBA, 0)
    {
        //
        // Local0 is the return package.  The status code is defaulted
        // to 'unknown failure' (0x07).
        //
        Store(Package(3){0x07,0x00,0x00}, Local0)

        //
        // Acquire the SMBus mutex to ensure transactional synchronization.
        //
        If(LEqual(Acquire(SBX1, 0xFFFF), 0))
        {
            //
            // Make sure there's an active (non-consumed) alarm by
            // checking bit 6 of the status register.  If not, the
            // return package already indicates that there isn't an
            // active alarm.
            //
            If(And(STS, 0x40))
            {
                Store(0x00, Index(Local0, 0))       // Status = Success

                ShiftRight(AADR, 1, Index(Local0, 1))       // Slave Address

                Store(2, Index(Local0, 2))          // Data Length

                Store(ShiftLeft(ADB1, 8), Local1)   // Data
                Add(Local1, ADB0, Local1)
                Store(Local1, Index(Local0, 3))

                //
                // Clear the alarm by resetting the status register.
                //
                Store(0x00, STS)
            }
            Else
            {
                Store(0x01, Index(Local0, 0))       // Status = No Alert
            }

            Release(SBX1)
        }

        Return(Local0)
    } // _SBA()

} // Device(SMB1)
```

## Maxim 1617

The Maxim 1617 ASIC includes two thermal sensors: a local sensor that monitors the temperature of the ASIC itself, and a remote sensor that on the mobile reference platform provides temperature readings from the system's Intel Pentium® II processor. This device is connected to the EC SMBus at slave address 0x4E (7-bit non-shifted value).

The ASIC includes an interrupt output line and supports the SMBus alert response message – making it capable of generating hardware alerts. But due to a design limitation, this capability cannot be used to generate an ACPI-visible interrupt (SCI) and thus does not support hardware alerting as defined in this specification.

A full listing of the ASL to implement this functionality is provided below.

```
////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
//                      Maxim 1617 Sensor Definitions                     //
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

//
// MSE (Multiply & Sign Extend):
// -----------------------------
// Utility method that converts values obtained from the Maxim 1617
// from degrees C to 1/10 degrees C and performs 32-bit sign extension
// for negative values.
//
Method(MSE, 1)
{
    Store(Arg0, Local0)
    If(And(Local0, 0x80))
    {
        Or(Local0,0xFFFFFF00,Local0)
    }
    Return(Multiply(Local0, 10))
} // MSE()

////////////////////////////////////////////////////////////////////////////
// Device:      TS00
// ------------------------------------------------------------------------
// Description: Remote (Processor) Temperature Sensor on Maxim 1617.
//              Note that state capabilities and thresholds are not exposed
//              since this sensor is used (controlled) by the OS in the
//              implementation of a thermal zone.
////////////////////////////////////////////////////////////////////////////
Device(TS00)
{
    Name(_HID, "MGMT101")
    Name(_UID, 0x01010100)                 // 1st Metolious Thermal Sensor

    //
    // INFO (Sensor Information):
    // --------------------------
    // Return a SENSOR_INFO structure describing the properties of
    // this sensor.
    //
    Method(INFO)
    {
        Return(Buffer()
        {
            0x01,                          // Object Type (Sensor)
            0x10,                          // Version (v1.0)
            0x01,                          // Major Sensor Type (Thermal)
            0x01,                          // Minor Sensor Type (Standard Thermal Sensor)
            0x00,                          // Sensor Instance (1st Instance)
            0x01,                          // Hardware Capability (Numeric)
            0x21,                          // Monitored Device Type (Processor Module)
            0x00,                          // Monitored Device Instance (1st Instance)
            0x04,                          // Property Count
            0x01, 0x00, 0x00, 0x01, 0x5E,  // Property: Nominal Reading (+35.0C)
            0x02, 0x00, 0x00, 0x04, 0xF6,  // Property: Physical Maximum Reading (+127.0 C)
            0x03, 0xFF, 0xFF, 0xFD, 0x76,  // Property: Physical Minimum Reading (-65.0 C)
            0x05, 0x00, 0x00, 0x00, 0x0A,  // Property: Resolution (1.0C)
        })
    } // INFO()

    //
    // SR (Sensor Reading):
    // --------------------
    // Return the current sensor reading.
    //
    Method(SR)
    {
        //
        // Set the return value to UNKNOWN, just in case...
        //
```

```
        Store(0x80000000, Local0)

        //
        // Read the temperature ('RRTE') and make sure the data is
        // valid.  Convert to 1/10th degree Celsius / sign extend.
        //
        Store(\_SB.PCI0.PX40.EC0.SMB1._SBR(0x07, 0x4E, 0x01), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 2)), Local0)
            Store(MSE(Local0), Local0)
        }

        Return(Local0)
    } // SR()

} // Device(TS00)

////////////////////////////////////////////////////////////////////////////
// Device:      TS01
// ------------------------------------------------------------------------
// Description: Local (Ambient) Temperature Sensor on Maxim 1617.
//              Note that although the Maxim 1617 supports HW alerting,
//              this feature is not exposed because the reference
//              hardware being used does not map the SMB Alert pin to
//              the EC-SMBus controller.
////////////////////////////////////////////////////////////////////////////
Device(TS01)
{
    Name(_HID, "MGMT101")
    Name(_UID, 0x01010101)                  // 2nd Metolious Thermal Sensor

    //
    // INIT (Sensor Initialization):
    // -----------------------------
    // Initialize this sensor.
    //
    Method(INIT)
    {
        //
        // Set the default thresholds (50.0C upper, 0.0C lower).
        //
        STC(0x04, 500)
        STC(0x08, 0)
    } // INIT()

    //
    // INFO (Sensor Information):
    // --------------------------
    // Return a SENSOR_INFO structure describing the properties of
    // this sensor.
    //
    Method(INFO)
    {
        Return(Buffer()
        {
            0x01,                           // Object Type (Sensor)
            0x10,                           // Version (v1.0)
            0x01,                           // Major Sensor Type (Thermal)
            0x01,                           // Minor Sensor Type (Standard Thermal Sensor)
            0x01,                           // Sensor Instance (2nd Instance)
            0x07,                           // Hardware Capability (State-based Numeric
                                            //                      sensor w/ Thresholds)
            0x10,                           // Monitored Device Type (System Board)
            0x00,                           // Monitored Device Instance (1st Instance)
            0x09,                           // Property Count
            0x01, 0x00, 0x00, 0x01, 0x5E,   // Property: Nominal Reading (+35.0C)
            0x02, 0x00, 0x00, 0x04, 0xF6,   // Property: Physical Maximum Reading (+127.0 C)
            0x03, 0xFF, 0xFF, 0xFD, 0x76,   // Property: Physical Minimum Reading (-65.0 C)
            0x05, 0x00, 0x00, 0x00, 0x0A,   // Property: Resolution (1.0C)
            0x10, 0x00, 0x00, 0x00, 0x05,   // Property: State Capability (OK & Critical States)
```

```
        0x11, 0x00, 0x00, 0x01, 0x2C,   // Property: Polling Frequency (5 minutes)
        0x20, 0x00, 0x00, 0x00, 0x0C,   // Property: Threshold Capability (Upper &
                                        //                   Lower Critical Threshold)
        0x23, 0x00, 0x00, 0x01, 0xF4,   // Property: Default Upper Critical Threshold (50.0C)
        0x24, 0x00, 0x00, 0x00, 0x00    // Property: Default Lower Critical Threshold (0.0C)
    })
} // INFO()

//
// SR (Sensor Reading):
// --------------------
// Return the current sensor reading.
//
Method(SR)
{
    //
    // Set the return value to UNKNOWN, just in case...
    //
    Store(0x80000000, Local0)

    //
    // Read the temperature ('RLTS') and make sure the data is
    // valid.  Convert to 1/10th degree Celsius / sign extend.
    //
    Store(\_SB.PCI0.PX40.EC0.SMB1._SBR(0x07, 0x4E, 0x00), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
        Store(Derefof(Index(Local1, 2)), Local0)
        Store(MSE(Local0), Local0)
    }

    Return(Local0)
} // SR()

//
// SS (Sensor State):
// ------------------
// Return the current sensor state.
//
Method(SS)
{
    //
    // Set the return value to UNKNOWN, just in case...
    //
    Store(0x80000000, Local0)

    //
    // Read the Maxim's status byte ('RSL'), check for a
    // collision (with internal A/D process), and see if
    // either the upper or lower critical thresholds have
    // been crossed.
    //
    Store(\_SB.PCI0.PX40.EC0.SMB1._SBR(0x07, 0x4E, 0x02), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
        Store(Derefof(Index(Local1, 1)), Local0)

        //
        // Make sure there hasn't been a collision.  Note
        // that if a collision occurs this control method
        // will return UNKNOWN.
        //
        If(LNotEqual(And(Local0, 0x7F), 0x7F))
        {
            If(And(Local0, 0x60))
            {
                Store(0x03, Local0)     // Status is CRITICAL.
            }
            Else
            {
                Store(0x01, Local0)     // Status is OK.
```

```
            }
        }
    }
} // SS()

//
// STQ (Sensor Threshold Query):
// ----------------------------
// Returns current threshold values.
//
Method(STQ, 1)
{
    //
    // Set the return value to UNKNOWN, just in case...
    //
    Store(0x80000000, Local0)

    //
    // Upper Critical Threshold?
    //
    if(And(Arg0, 0x04))
    {
        //
        // Read the threshold ('RLHN') and make sure the data is
        // valid.  Convert to 1/10th degree Celsius / sign extend.
        //
        Store(\_SB.PCI0.PX40.EC0.SMB1._SBR(0x07, 0x4E, 0x05), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 1)), Local0)
            Store(MSE(Local0), Local0)
        }
    }

    //
    // Lower Critical Threshold?
    //
    if(And(Arg0, 0x08))
    {
        //
        // Read the threshold ('RLLI') and make sure the data is
        // valid.  Convert to 1/10th degree Celsius / sign extend.
        //
        Store(\_SB.PCI0.PX40.EC0.SMB1._SBR(0x07, 0x4E, 0x06), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 1)), Local0)
            Store(MSE(Local0), Local0)
        }
    }

    Return(Local0)

} // STQ()

//
// STC (Sensor Threshold Control):
// ------------------------------
// Update threshold values.
//
Method(STC, 2)
{
    //
    // Convert threshold value from 1/10th to degrees C.
    //
    If(LNotEqual(Arg1, 0))
    {
        Divide(Arg1, 10, Local1, Local0)
    }
    Else
    {
```

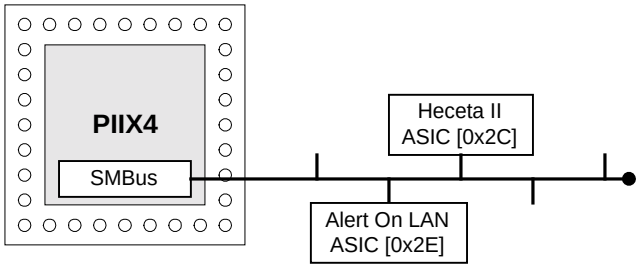
```
            Store(0, Local0)
        }

        //
        // Upper Critical Threshold? ('WLHO')
        //
        if(And(Arg0, 0x04))
        {
            \_SB.PCI0.PX40.EC0.SMB1._SBW(0x06, 0x4E, 0x0B, 0x01, Local0, 0x00)
        }

        //
        // Lower Critical Threshold? ('WLLM')
        //
        if(And(Arg0, 0x08))
        {
            \_SB.PCI0.PX40.EC0.SMB1._SBW(0x06, 0x4E, 0x0C, 0x01, Local0, 0x00)
        }

    } // STC()

} // Device(TS01)
```

# A.2 Desktop Example

## A.2.1 Overview

The sample ASL presented in this section is based on a desktop example system that includes a Heceta II ASIC (providing thermal, fan, voltage, and chassis intrusion sensors) and Alert on LAN* ASIC (providing a system hang watchdog) that are connected to the system using a PIIX4 SMBus. Figure 13 illustrates the platform manageability hardware configuration for this system.

**Figure 13:** Desktop Example System




Information on the PIIX4 is available at:
http://developer.intel.com/design/intarch/DATASHTS/209562.htm

Information on the SMBus is available at:
http://www.sbs-forum.org/smbus/specs/

A functional overview of the Heceta II is available at:
http://developer.intel.com/ial/wfm/wfm20/design/sensdt/HEC2FUNC.HTM

Examples of Heceta II-compliant ASICs include the Analog Devices ADM9240* and Dallas Semiconductor DS1780*. Information on these devices is available at:
http://products.analog.com/products/info.asp?product=adm9240
http://www.dalsemi.com/DocControl/PDFs/1780.pdf

Information on the Alert on LAN ASIC is available at:
http://developer.intel.com/design/network/datashts/69281801.pdf

## A.2.2 PIIX4 SMBus

Implementing PIIX4 SMBus access from AML is required to communicate with the Heceta II ASIC and successfully model the sensors it contains. For simplicity, the PIIX4 SMBus control methods have been added to the root of the system bus (_SB) hierarchy in the ACPI namespace. Ideally this object should exist under the bus that it is connected to (e.g. ISA0) – as shown in Figure 1.

As discussed in section 3.3, access to the SMBus must be coordinated between the various SMBus consumers. To facilitate this, the interface defined in the *SMBus Control Method Interface Specification* has been employed in this sample ASL to provide synchronization between AML and OS software. Developers should obtain the latest version of the SMBus CMI specification and model their implementation accordingly.

For this example we'll assume that the PIIX4 SMBus registers reside at offset 0x7000 in system I/O space. A full listing of the ASL to implement this functionality is provided below.

```
////////////////////////////////////////////////////////////////////////////
// Device:      SMB0
// ------------------------------------------------------------------------
// Description: The PIIX4 SMBus segment (includes the ADM9240 and AOL ASIC).
//              This ASL assumes the SMBus is enabled and its registers are
//              located at offset 0x7000.
////////////////////////////////////////////////////////////////////////////
Device(SMB0)
{
    Name(_HID, "SMBUS01")
    Name(_UID, 0)

    Mutex(SBX0, 0)          // SMBus transactional synchronization.

    //
    // OperationRegion:
    // ----------------
    // The PIIX4 SMBus registers reside at offset 0x7000 in system IO space.
    //
    OperationRegion(SMB0,SystemIO,0x7000,0x0C)
    Field(SMB0,ByteAcc,NoLock,Preserve)
    {
        HSTS,   8,  // Host Status
        SSTS,   8,  // Slave Status
        HCNT,   8,  // Host Control
        HCMD,   8,  // Host Command
        HADD,   8,  // Host Address
        DAT0,   8,  // Host Data Byte 0
        DAT1,   8,  // Host Data Byte 1
        BLKD,   8,  // Host Block Data
        SCNT,   8,  // Slave Count
        SCMD,   8,  // Shadow Command
        SEVT,   8,  // Slave Event
        SDAT,   8   // Slave Data

    }

    //
    // _SBI (SMBus Information):
    // -------------------------
    // Returns a SMB_INFO structure describing the properties of this
    // SMBus segment.
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = SMBus Control Method Interface (CMI) version (Integer)
    //  (1)     = SMB_INFO data structure (Buffer)
    //
    Method(_SBI)
    {
        //
        // Local0 is the return package.  The CMI version is set to v1.0 (0x10)
        //
        Store(Package(2){0x10,0x00}, Local0)

        Store(Buffer()
        {
            0x10,                   // SMB_INFO structure  version (v1.0)
            0x10,                   // SMBus Specification Version (v1.0)
            0x00,                   // Segment Hardware Capability
            0x00,                   // Alert Poling Interval (no alert capable devices)
            0x02,                   // Device Count
            0x2C, 0x00,             // SMBus Device #1: ADM9240
            0x00, 0x00, 0x11, 0xD4, //      SMB_UDID... (Device ID is a placeholder)
            0x00, 0x01, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00,
            0x00, 0x00, 0x00, 0x00,
            0x2E, 0x00,             // SMBus Device #2: AOL ASIC
            0x00, 0x00, 0x80, 0x86, //      SMB_UDID... (Device ID is a placeholder)
```

```
        0x00, 0x04, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x00, 0x00,
    }, Index(Local0, 1))
    Return(Local0)
} // _SBI

//
// SWTC (Wait for Transaction Complete):
// -------------------------------------
// Wait until the previous SMBus transaction has completed.
//
// Parameters:
//  Arg0    = Timeout Value (in ms)
//
// Return Value:
//  0x00    = OK
//  0x07    = Unknown Failure
//  0x10    = Address Not Acknowledged
//  0x11    = Device Error
//  0x12    = Command Access Denied
//  0x13    = Unknown Error
//  0x17    = Device Access Denied
//  0x18    = Timeout
//  0x19    = Unsupported Protocol
//  0x1A    = Bus Busy
//  0x1F    = PEC (CRC-8) Error
//
Method(SWTC, 1)
{
    Store(Arg0, Local0)
    Store(0x07, Local2)

    //
    // The previous command has completed when bit 1 ('interrupt status')
    // of the host status (HSTS) register is set or an error occurs.
    // Wait <timeout> ms (in 10ms chunks) for this to occur.
    //
    Store(1, Local1)
    While(LEqual(Local1, 1))
    {
        //
        // Read the hosts status (HSTS) register, mask off bits 4:1, and
        // check to see if this transaction has completed.  Note that the
        // transaction is being processed while these bits are non-zero.
        //
        Store(And(HSTS, 0x1E), Local3)
        If(LNotEqual(Local3, 0))
        {
            //
            // See if this transaction was successful.  Note that errors
            // are reported in bits 4:2.
            //
            If(LEqual(Local3, 0x02))
            {
                Store(0x00, Local2)     // Success.
            }
            Else
            {
                Store(0x07, Local2)     // ERROR: Unknown Error.
            }

            Store(0, Local1)            // Terminate loop.
        }
        Else
        {
           //
           // Transaction isn't complete.  Check for timeout, and if not,
            // sleep 10ms and loop again.
           //
           If(LLess(Local0, 10))
```

```
            {
                Store(0x18, Local2)             // ERROR: Timeout occurred.
                Store(0x00, Local1)             // Terminate loop.
            }
            Else
            {
                Sleep(10)
                Subtract(Local0, 10, Local0)
            }
        }
    }

    Return(Local2)
} // Method(SWTC)

//
// _SBR (SMBus Read):
// ------------------
//
// Parameters:
//  Arg0    = Protocol Value (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command Code (Integer)
//
// Return Value (Package):
//  (0)     = Status Code (Integer)
//  (1)     = Data Length (Integer)
//  (2)     = Data (Integer | Buffer)
//
Method(_SBR, 3)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(3){0x07,0x00,0x00}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking.
    //
    If(LNotEqual(Arg0, 0x03))                   // Read Quick
    {
        If(LNotEqual(Arg0, 0x05))               // Receive Byte
        {
            If(LNotEqual(Arg0, 0x07))           // Read Byte
            {
                If(LNotEqual(Arg0, 0x09))       // Read Word
                {
                    If(LNotEqual(Arg0, 0x0B))   // Read Block
                    {
                        Store(0x19, Index(Local0, 0))
                        Return(Local0)
                    }
                }
            }
        }
    }

    //
    // Acquire the SMBus mutex to ensure transactional
    // synchronization.
    //
    If(LEqual(Acquire(SBX0, 0xFFFF), 0))
    {
        //
        // Translate the slave address (shift left + set 'read' bit) and
        // write this and the command byte to the SMBus registers.  Clear
        // the host status register (preserving bits 7:5).
        //
```

```
        Store(Or(ShiftLeft(Arg1, 0x01), 0x01), HADD)
        Store(Arg2, HCMD)
        Store(Or(HSTS, 0x1F), HSTS)

        //
        // Specify the protocol using bits 4:2 of the host control
        // register (preserving bits 5 & 7) and start the transaction
        // by writing a '1' to bit 6 of the host control register.
        //
        If(LEqual(Arg0, 0x03))          // Read Quick
        {
            Store(Or(And(HCNT, 0xA0), 0x40), HCNT)
        }
        If(LEqual(Arg0, 0x05))          // Receive Byte
        {
            Store(Or(And(HCNT, 0xA0), 0x44), HCNT)
        }
        If(LEqual(Arg0, 0x07))          // Read Byte
        {
            Store(Or(And(HCNT, 0xA0), 0x48), HCNT)
        }
        If(LEqual(Arg0, 0x09))          // Read Word
        {
            Store(Or(And(HCNT, 0xA0), 0x4C), HCNT)
        }
        If(LEqual(Arg0, 0x0B))          // Read Block
        {
            Store(Or(And(HCNT, 0xA0), 0x54), HCNT)
        }

        //
        // Wait (up to 1 second) for the transaction to complete.  Store
        // the status code and, if successful, the data length and data
        // into the return package.
        //
        Store(SWTC(1000), Local1)
        Store(Local1, Index(Local0, 0))

        If(LEqual(Local1, 0))
        {
            If(LEqual(Arg0, 0x05))          // Receive Byte
            {
                Store(1, Index(Local0, 1))
                Store(DAT0, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x07))          // Read Byte
            {
                Store(1, Index(Local0, 1))
                Store(DAT0, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x09))          // Read Word
            {
                Store(2, Index(Local0, 1))
                Store(DAT1, Local2)
                ShiftLeft(Local2, 8, Local2)
                Add(Local2, DAT0, Local2)
                Store(Local2, Index(Local0, 2))
            }
            If(LEqual(Arg0, 0x0B))          // Read Block
            {
                // TBD...
            }
        }

        Release(SBX0)
    }

    Return(Local0)
} // _SBR()

//
```

```
// _SBW (SMBus Write):
// ------------------
//
// Parameters:
//  Arg0    = Protocol Value (Integer)
//  Arg1    = Slave Address (Integer)
//  Arg2    = Command Code (Integer)
//  Arg3    = Data Length (Integer)
//  Arg4    = Data (Integer | Buffer)
//
// Return Value (Package):
//  (0)     = Status (Integer)
//
Method(_SBW, 5)
{
    //
    // Local0 is the return package.  The status code is defaulted
    // to 'unknown failure' (0x07).
    //
    Store(Package(1){0x07}, Local0)

    //
    // Make sure the protocol is valid, if not return the
    // 'invalid protocol' status code.  Note that this segment
    // does not support packet error checking or the 'process call'
    // protocol.
    //
    If(LNotEqual(Arg0, 0x02))                   // Write Quick
    {
        If(LNotEqual(Arg0, 0x04))               // Send Byte
        {
            If(LNotEqual(Arg0, 0x06))           // Write Byte
            {
                If(LNotEqual(Arg0, 0x08))       // Write Word
                {
                    If(LNotEqual(Arg0, 0x0A))   // Write Block
                    {
                        Store(0x19, Index(Local0, 0))
                        Return(Local0)
                    }
                }
            }
        }
    }

    //
    // Acquire the SMBus mutex to ensure transactional synchronization.
    //
    If(LEqual(Acquire(SBX0, 0xFFFF), 0))
    {
        //
        // Translate the slave address (shift left) and write this and
        // the command byte to the SMBus registers.  Clear the host status
        // register (preserving bits 7:5).
        //
        Store(ShiftLeft(Arg1, 0x01), HADD)
        Store(Arg2, HCMD)
        Store(Or(HSTS, 0x1F), HSTS)

        //
        // Store the data to be written (if any), specify the protocol
        // using bits 4:2 of the host control register, and start the
        // transaction by writing a '1' to bit 6 of the host control
        // register.
        //
        If(LEqual(Arg0, 0x02))          // Write Quick
        {
            Store(Or(And(HCNT, 0xA0), 0x40), HCNT)
        }
        If(LEqual(Arg0, 0x04))          // Send Byte
        {
```

```
                Store(Or(And(HCNT, 0xA0), 0x44), HCNT)
            }
            If(LEqual(Arg0, 0x06))          // Write Byte
            {
                Store(Arg4, DAT0)
                Store(Or(And(HCNT, 0xA0), 0x48), HCNT)
            }
            If(LEqual(Arg0, 0x08))          // Write Word
            {
                And(Arg4, 0x00FF, DAT0)
                ShiftRight(Arg4, 8, DAT1)
                Store(Or(And(HCNT, 0xA0), 0x4C), HCNT)
            }
            If(LEqual(Arg0, 0x0A))          // Write Block
            {
                // TBD...
                Store(Or(And(HCNT, 0xA0), 0x54), HCNT)
            }

            //
            // Wait (up to 1 second) for the transaction to complete.  Store
            // the status code into the return package.
            //
            Store(SWTC(1000), Index(Local0, 0))

            Release(SBX0)
        }

        Return(Local0)
    } // _SBW()

    //
    // _SBA (SMBus Alert Information):
    // -------------------------------
    //
    // Parameters:
    //  <none>
    //
    // Return Value (Package):
    //  (0)     = Status Code (Integer) -> {0x00=Success | 0x01=No Active Alert
    //                                      | 0x07=Unknown Failure}
    //  (1)     = Slave Address (Integer)
    //  (2)     = Data Length (Integer)
    //  (3)     = Data (Integer)
    //
    Method(_SBA, 0)
    {
        //
        // Local0 is the return package.  The status code is defaulted
        // to 'unknown failure' (0x07).
        //
        Store(Package(3){0x07,0x00,0x00}, Local0)

        //
        // Acquire the SMBus mutex to ensure transactional synchronization.
        //
        If(LEqual(Acquire(SBX0, 0xFFFF), 0))
        {
            // TBD...
            Release(SBX0)
        }

        Return(Local0)
    } // _SBA()

} // Device(SMB0)
```

### A.2.3 Heceta II

The Heceta II ASIC includes the following sensors:

- One thermal sensor.
- Two fan speed sensors.
- Six voltage sensors.
- One chassis intrusion sensor.

Heceta II ASICs include an interrupt output line but the device cannot generate SMBus alerts due to the lack of support for the SMBus alert response message. This prevents the Heceta II from being able to generate ACPI-visible (and thus Metolious-compliant) hardware alerts. Additionally, the Interrupt Status Registers (0x41 and 0x42) cannot be used to detect sensor state due to the automatic clearing of these registers following each read. Because of this the state must be computed by manually comparing the sensor reading against the current threshold(s). This situation also prevents the use of the temperature hysteresis (register 0x3A).

For simplicity, the sensor objects in this example have been added to the root of the system bus (_SB) hierarchy in the ACPI namespace. Ideally these objects should exist under the bus that they are connected to (e.g. SMB0) – as shown in Figure 1.

For this example we'll assume that the Heceta II is connected to the PIIX4 SMBus at slave address 0x2C (7-bit non-shifted value). A full listing of the ASL to implement this functionality is provided below.

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//                        Heceta II Sensor Definitions                    //
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Device:      TS00
// ------------------------------------------------------------------------
// Description: Local (Ambient) Temperature Sensor on Heceta II.
////////////////////////////////////////////////////////////////////////////
Device(TS00)
{
    Name(_HID, "MGMT101")
    Name(_UID, 0x01010100)                 // Sensor Signature (Thermal Sensor)

    //
    // _INI (Device Initialization):
    // -----------------------------
    // This control method, executed by ACPI immediately after the OS loads,
    // is used to initialize the Heceta II ASIC.  We enable the ASIC's
    // environmental monitoring by writing a '1' to the START bit of the
    // Configuration Register.  Note that this only needs to occur once, thus
    // only exists in the TS00 object.
    //
    Method(_INI)
    {
        \_SB.SMB0._SBW(0x06, 0x2C, 0x40, 0x01, 0x01, 0x00)
    } // _INI()

    //
    // INIT (Sensor Initialization):
    // -----------------------------
    // Initialize this sensor.
    //
    Method(INIT)
    {
        //
        // Set the Temperature Configuration Register to "Comparator Mode".
        // This disables hysterisis (which cannot be used anyway because the
        // device does not fully support hardware alerting), and initialize
        // the device with default upper critical threshold (35.0C).
        //
        \_SB.SMB0._SBW(0x06, 0x2C, 0x4B, 0x01, 0x03, 0x00)
```

```
    STC(0x04, 350)
} // INIT()

//
// INFO (Sensor Information):
// --------------------------
// Return a SENSOR_INFO structure describing the properties of this sensor.
//
Method(INFO)
{
    Return(Buffer() {
        0x01,                           // Object Type (Sensor)
        0x10,                           // Version (v1.0)
        0x01,                           // Major Sensor Type (Thermal)
        0x01,                           // Minor Sensor Type (Standard Thermal Sensor)
        0x00,                           // Sensor Instance (1st Instance)
        0x07,                           // Hardware Capability (State-based Numeric
                                        //                 sensor w/ Thresholds)
        0x11,                           // Monitored Device Type (Motherboard)
        0x00,                           // Monitored Device Instance (1st Instance)
        0x08,                           // Property Count
        0x01, 0x00, 0x00, 0x01, 0x18,   // Property: Nominal Reading (+28.0C)
        0x02, 0x00, 0x00, 0x04, 0xE2,   // Property: Physical Maximum Reading (+125.0 C)
        0x03, 0xFF, 0xFF, 0xFE, 0x70,   // Property: Physical Minimum Reading (-40.0 C)
        0x05, 0x00, 0x00, 0x00, 0x05,   // Property: Resolution (0.5C)
        0x10, 0x00, 0x00, 0x00, 0x05,   // Property: State Capability (OK & Critical States)
        0x11, 0x00, 0x00, 0x01, 0x2C,   // Property: Polling Frequency (5 minutes)
        0x20, 0x00, 0x00, 0x00, 0x04,   // Property: Threshold Capability (Upper
                                        //                           Critical Threshold)
        0x23, 0x00, 0x00, 0x01, 0x5E    // Property: Default Upper Critical Threshold (35.0C)
    })
} // INFO()

//
// Sensor Reading (SR):
// --------------------
// Return the current sensor reading.
//
Method(SR)
{
    //
    // Set the return value to UNKNOWN.
    //
    Store(0x80000000, Local0)

    //
    // Read the temperature's MSBs and make sure the data is valid.
    //
    Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x27), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
        Store(Derefof(Index(Local1, 2)), Local0)

        //
        // Negative temperatures need to be sign-extended (later).
        //
        And(0x80, Local0, Local2)

        //
        // Convert value to 1/10 degrees C.
        //
        Multiply(Local0, 10, Local0)

        //
        // Read the temperature's LSB (0.5C) and make sure the data is valid.
        //
        Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x4B), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            //
            // Add 0.5C if high-order bit of TCR register is set.
```

```
            //
            if (And(Derefof(Index(Local1, 2)), 0x80))
            {
                Add(Local0, 5, Local0)
            }
        }

        //
        // Sign extend if negative temperature.
        //
        if (Local2)
        {
            Subtract(0, Local0, Local0)
        }
    }

    Return (Local0)
} // SR()

//
// Sensor State (SS):
// ------------------
// Heceta II ASICs include status registers but these cannot be used
// due to the automatic clearing of these register after any read
// operation.  Instead, we must get the sensor reading and manually
// compare against the current thresholds.
//
Method(SS)
{
    //
    // Get the current upper critical threshold.  If the threshold is
    // unknown, return the UNKNOWN state.  If the threshold is disabled,
    // return the OK state.
    //
    Store(STQ(0x04), Local1)
    If(LEqual(Local1, 0x80000000))
    {
        Return(0x80000000)  // UNKNOWN state.
    }
    If(LEqual(Local1, 0x7FFFFFFF))
    {
        Return(0x01)        // OK state.
    }

    //
    // Get the current sensor reading and make sure it is valid. If the
    // reading is unknown, return the UNKNOWN state.
    //
    Store(SR, Local2)
    If(LEqual(Local1, 0x80000000))
    {
        Return(0x80000000)  // UNKNOWN state.
    }

    //
    // Compare the current reading and threshold values.
    //
    If (LGreaterEqual(Local2, Local1))
    {
        Return(0x04)        // CRITICAL state.
    }
    Else
    {
        Return(0x01)        // OK state.
    }
} // SS()

//
// Sensor Threshold Query (STQ):
// -----------------------------
// Returns current threshold values.
```

```
    //
    Method(STQ, 1)
    {
        //
        // Set the return value to UNKNOWN.
        //
        Store(0x80000000, Local0)

        //
        // This sensor only supports an upper critical threshold.
        //
        If(And(Arg0, 0x04))
        {
            //
            // Get the current upper critical threshold value and make
            // sure the data is valid.
            //
            Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x39), Local1)
            If(LEqual(Derefof(Index(Local1, 0)), 0))
            {
                Store(Derefof(Index(Local1, 2)), Local0)

                //
                // If the threshold is disabled, return the DISABLED
                // value.  Otherwise convert to 1/10 degree C.
                //
                if (LEqual(Local0, 0x7F))
                {
                    Return(0x7FFFFFFF)  // Threshold is DISABLED.
                }
                Else
                {
                    Multiply(Local0, 10, Local0)
                }
            }
        }

        Return(Local0)
    } // STQ()

    //
    // Sensor Threshold Control (STC):
    // -------------------------------
    // Updates threshold values.
    //
    Method(STC, 2)
    {
        //
        // This sensor only supports an upper critical threshold.
        //
        If(And(Arg0, 0x04))
        {
            //
            // If this threshold should be disable write the value 0x7F
            // (the maximum temperature) to the threshold register.
            // Otherwise convert the value to degrees C and store.
            //
            If(LEqual(Arg1, 0x7FFFFFFF))
            {
                Store(0x7F, Local0)
            }
            Else
            {
                Divide(Arg1, 10, Local1, Local0)
            }

            \_SB.SMB0._SBW(0x06, 0x2C, 0x39, 0x01, Local0, 0x00)
        }
    } // STC()

} // Device(TS00)
```

```
////////////////////////////////////////////////////////////////////////////
// Device:      FS00
// ------------------------------------------------------------------------
// Description: Primary Fan Sensor on Heceta II.
////////////////////////////////////////////////////////////////////////////
Device(FS00)
{
    Name(_HID, "MGMT102")
    Name(_UID, 0x01020100)                // Sensor Signature (1st Fan Sensor)

    //
    // _STA (Device Status):
    // ---------------------
    // Detect if a fan is connected to this sensor by checking the current
    // sensor reading.  If no fan is present, this device will not be
    // enabled/enumerated by the PnP subsystem.
    //
    Method(_STA)
    {
        //
        // Get the current sensor reading and make sure it is valid.
        // If not, return 0 (zero) indicate that the device is not
        // present.  Otherwise return 0x0F.
        //
        Store(SR, Local0)

        If(LEqual(Local0, 0x00))  // Fan not present (or stopped).
        {
            Return(0x00)
        }
        Else
        {
            Return(0x0F)          // Device present & working fine.
        }
    }

    //
    // INIT (Sensor Initialization):
    // -----------------------------
    // Initialize this sensor.
    //
    Method(INIT)
    {
        //
        // Set the fan divisor to 2 (ASIC default) and set the default
        // upper critical threshold to 70% of this nominal value (3080 RPM).
        //
        \_SB.SMB0._SBW(0x06, 0x2C, 0x47, 0x01, 0x50, 0x00)
        STC(0x08, 3080)
    } // INIT()

    //
    // INFO (Sensor Information):
    // --------------------------
    // Return a SENSOR_INFO structure describing the properties of this sensor.
    //
    Method(INFO)
    {
        Return(Buffer() {
            0x01,                         // Object Type (Sensor)
            0x10,                         // Version (v1.0)
            0x02,                         // Major Sensor Type (Cooling Device)
            0x01,                         // Minor Sensor Type (Standard Fan Sensor)
            0x00,                         // Sensor Instance (1st Instance)
            0x07,                         // Hardware Capability (State-based Numeric
                                          //    sensor w/ Thresholds)
            0x22,                         // Monitored Device Type (Processor Cartridge)
            0x00,                         // Monitored Device Instance (1st Instance)
            0x08,                         // Property Count
            0x01, 0x00, 0x00, 0x11, 0x30,   // Property: Nominal Reading (4400 RPM)
```

```
            0x02, 0x80, 0x00, 0x00, 0x00,   // Property: Physical Maximum Reading (??? RPM)
            0x03, 0x80, 0x00, 0x00, 0x00,   // Property: Physical Minimum Reading (??? RPM)
            0x05, 0x80, 0x00, 0x00, 0x00,   // Property: Resolution (??? RPM)
            0x10, 0x00, 0x00, 0x00, 0x05,   // Property: State Capability (OK & Critical States)
            0x11, 0x00, 0x00, 0x01, 0x2C,   // Property: Polling Frequency (5 minutes)
            0x20, 0x00, 0x00, 0x00, 0x08,   // Property: Threshold Capability (Lower
                                            //               Critical Threshold)
            0x24, 0x00, 0x00, 0x0C, 0x08    // Property: Default Lower Critical
                                            //                         Threshold (3080 RPM)
        })
    } // INFO()

    //
    // Sensor Reading (SR):
    // --------------------
    // Return the current sensor reading.
    //
    Method(SR)
    {
        //
        // Set the return value to UNKNOWN.
        //
        Store(0x80000000, Local0)

        //
        // Read the fan count and make sure the data is valid.
        //
        Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x28), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 2)), Local0)

            //
            // A count value of 0xFF indicates that the fan is stopped;
            // return 0 RPMs.  Otherwise convert 'count' to RPMs.
            //
            If(LEqual(Local0, 0xFF))
            {
                Store(0x00, Local0)
            }
            Else
            {
                //
                // The fan RPM formula is as follows (see the Heceta II spec):
                //  RPM = 1350000/(divisor*count)
                //
                if (LGreater(Local0, 0))
                {
                    Divide(675000, Local0, Local1, Local0)
                }
                Else
                {
                    Store(0x80000000, Local0)
                }
            }

        }

        Return (Local0)
    } // SR()

    //
    // Sensor State (SS):
    // ------------------
    // Heceta II ASICs include status registers but these cannot be used
    // due to the automatic clearing of these register after any read
    // operation.  Instead, we must get the sensor reading and manually
    // compare against the current thresholds.
    //
    Method(SS)
    {
```

```
    //
    // Get the current lower critical threshold.  If the threshold is
    // unknown, return the UNKNOWN state.  If the threshold is disabled,
    // return the OK state.
    //
    Store(STQ(0x08), Local1)
    If(LEqual(Local1, 0x80000000))
    {
        Return(0x80000000)  // UNKNOWN state.
    }
    If(LEqual(Local1, 0x7FFFFFFF))
    {
        Return(0x01)        // OK state.
    }

    //
    // Get the current sensor reading and make sure it is valid. If the
    // reading is unknown, return the UNKNOWN state.
    //
    Store(SR, Local2)
    If(LEqual(Local1, 0x80000000))
    {
        Return(0x80000000)  // UNKNOWN state.
    }

    //
    // Compare the current reading and threshold values.
    //
    If (LLessEqual(Local2, Local1))
    {
        Return(0x04)        // CRITICAL state.
    }
    Else
    {
        Return(0x01)        // OK state.
    }
} // SS()

//
// Sensor Threshold Query (STQ):
// -----------------------------
// Returns current threshold values.
//
Method(STQ, 1)
{
    //
    // Set the return value to UNKNOWN.
    //
    Store(0x80000000, Local0)

    //
    // This sensor only supports a lower critical threshold.
    //
    If(And(Arg0, 0x08))
    {
        //
        // Get the current lower critical threshold value and make
        // sure the data is valid.
        //
        Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x3B), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 2)), Local0)

            //
            // Convert the threshold value from 'count' to RPM.  The
            // fan RPM formula is as follows (see the Heceta II spec):
            //  RPM = 1350000/(divisor*count)
            // Note that a threshold value of 0 (zero) indicates that
            // this threshold is disabled.
            //
```

```
                If (LEqual(Local0, 0))
                {
                    Store(0x7FFFFFFF, Local0)  // Threshold is DISABLED.
                }
                Else
                {
                    Divide(675000, Local0, Local1, Local0)
                }
            }
        }

        Return(Local0)
    } // STQ()

    //
    // Sensor Threshold Control (STC):
    // -------------------------------
    // Updates threshold values.
    //
    Method(STC, 2)
    {
        //
        // This sensor only supports a lower critical threshold.
        //
        If(And(Arg0, 0x08))
        {
            //
            // If this threshold should be disable write the value
            // 0 (zero) to the threshold register.  Otherwise convert
            // the value from RPM to 'count' and store.
            //
            If(LEqual(Arg1, 0))
            {
                Store(0, Local0)
            }
            Else
            {
                If(LEqual(Arg1, 0x7FFFFFFF))
                {
                    Store(0, Local0)
                }
                Else
                {
                    Divide(675000, Arg1, Local1, Local0)
                }
            }

            \_SB.SMB0._SBW(0x06, 0x2C, 0x3B, 0x01, Local0, 0x00)
        }
    } // STC()

} // Device(FS00)

////////////////////////////////////////////////////////////////////////////
// Device:      FS01
// ------------------------------------------------------------------------
// Description: Secondary Fan Sensor on Heceta II.
////////////////////////////////////////////////////////////////////////////
Device(FS01)
{
    Name(_HID, "MGMT102")
    Name(_UID, 0x01020101)                 // Sensor Signature (2nd Fan Sensor)

    //
    // _STA (Device Status):
    // ---------------------
    // Detect if a fan is connected to this sensor by checking the current
    // sensor reading.  If no fan is present, this device will not be
    // enabled/enumerated by the PnP subsystem.
    //
    Method(_STA)
```

```
{
    //
    // Get the current sensor reading and make sure it is valid.
    // If not, return 0 (zero) to indicate that the device is not
    // present.  Otherwise return 0x0F
    //
    Store(SR, Local0)

    If(LEqual(Local0, 0x00))  // Fan not present (or stopped).
    {
        Return(0x00)
    }
    Else
    {
        Return(0x0F)          // Device present & working fine.
    }
}

//
// INIT (Sensor Initialization):
// ----------------------------
// Initialize this sensor.
//
Method(INIT)
{
    //
    // Set the fan divisor to 2 (ASIC default), which implies that both
    // fans are running at a nominal speed at 4400 RPM, and set the default
    // upper critical threshold to 70% of this nominal value (3080 RPM).
    //
    \_SB.SMB0._SBW(0x06, 0x2C, 0x47, 0x01, 0x50, 0x00)
    STC(0x08, 3080)
} // INIT()

//
// INFO (Sensor Information):
// -------------------------
// Return a SENSOR_INFO structure describing the properties of this sensor.
//
Method(INFO)
{
    Return(Buffer() {
        0x01,                           // Object Type (Sensor)
        0x10,                           // Version (v1.0)
        0x02,                           // Major Sensor Type (Cooling Device)
        0x01,                           // Minor Sensor Type (Standard Fan Sensor)
        0x01,                           // Sensor Instance (2nd Instance)
        0x07,                           // Hardware Capability (State-based Numeric
                                        //            sensor w/ Thresholds)
        0x30,                           // Monitored Device Type (Power Source)
        0x00,                           // Monitored Device Instance (1st Instance)
        0x08,                           // Property Count
        0x01, 0x00, 0x00, 0x11, 0x30,   // Property: Nominal Reading (4400 RPM)
        0x02, 0x80, 0x00, 0x00, 0x00,   // Property: Physical Maximum Reading (??? RPM)
        0x03, 0x80, 0x00, 0x00, 0x00,   // Property: Physical Minimum Reading (??? RPM)
        0x05, 0x80, 0x00, 0x00, 0x00,   // Property: Resolution (??? RPM)
        0x10, 0x00, 0x00, 0x00, 0x05,   // Property: State Capability (OK & Critical States)
        0x11, 0x00, 0x00, 0x01, 0x2C,   // Property: Polling Frequency (5 minutes)
        0x20, 0x00, 0x00, 0x00, 0x08,   // Property: Threshold Capability (Lower
                                        //                   Critical Threshold)
        0x23, 0x00, 0x00, 0x0C, 0x08    // Property: Default Upper Critical
                                        //                        Threshold (3080 RPM)
    })
} // INFO()

//
// Sensor Reading (SR):
// --------------------
// Return the current sensor reading.
//
Method(SR)
```

```
{
    //
    // Set the return value to UNKNOWN.
    //
    Store(0x80000000, Local0)

    //
    // Read the fan count and make sure the data is valid.
    //
    Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x29), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
        Store(Derefof(Index(Local1, 2)), Local0)

        //
        // A count value of 0xFF indicates that the fan is stopped;
        // return 0 RPMs.  Otherwise convert 'count' to RPMs.
        //
        If(LEqual(Local0, 0xFF))
        {
            Store(0x00, Local0)
        }
        Else
        {
            //
            // The fan RPM formula is as follows (see the Heceta II spec):
            //  RPM = 1350000/(divisor*count)
            //
            if (LGreater(Local0, 0))
            {
                Divide(675000, Local0, Local1, Local0)
            }
            Else
            {
                Store(0x80000000, Local0)
            }
        }

    }

    Return (Local0)
} // SR()

//
// Sensor State (SS):
// ------------------
// Heceta II ASICs include status registers but these cannot be used
// due to the automatic clearing of these register after any read
// operation.  Instead, we must get the sensor reading and manually
// compare against the current thresholds.
//
Method(SS)
{
    //
    // Get the current lower critical threshold.  If the threshold is
    // unknown, return the UNKNOWN state.  If the threshold is disabled,
    // return the OK state.
    //
    Store(STQ(0x08), Local1)
    If(LEqual(Local1, 0x80000000))
    {
        Return(0x80000000)  // UNKNOWN state.
    }
    If(LEqual(Local1, 0x7FFFFFFF))
    {
        Return(0x01)        // OK state.
    }

    //
    // Get the current sensor reading and make sure it is valid. If the
    // reading is unknown, return the UNKNOWN state.
```

```
    //
    Store(SR, Local2)
    If(LEqual(Local1, 0x80000000))
    {
        Return(0x80000000)  // UNKNOWN state.
    }

    //
    // Compare the current reading and threshold values.
    //
    If (LLessEqual(Local2, Local1))
    {
        Return(0x04)        // CRITICAL state.
    }
    Else
    {
        Return(0x01)        // OK state.
    }
} // SS()

//
// Sensor Threshold Query (STQ):
// -----------------------------
// Returns current threshold values.
//
Method(STQ, 1)
{
    //
    // Set the return value to UNKNOWN.
    //
    Store(0x80000000, Local0)

    //
    // This sensor only supports a lower critical threshold.
    //
    If(And(Arg0, 0x08))
    {
        //
        // Get the current lower critical threshold value and make
        // sure the data is valid.
        //
        Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x3C), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 2)), Local0)

            //
            // Convert the threshold value from 'count' to RPM.  The
            // fan RPM formula is as follows (see the Heceta II spec):
            //  RPM = 1350000/(divisor*count)
            // Note that a threshold value of 0 (zero) indicates that
            // this threshold is disabled.
            //
            If (LGreater(Local0, 0))
            {
                Divide(675000, Local0, Local1, Local0)
            }
            Else
            {
                Store(0x7FFFFFFF, Local0)  // Threshold is DISABLED.
            }
        }
    }

    Return(Local0)
} // STQ()

//
// Sensor Threshold Control (STC):
// -------------------------------
// Updates threshold values.
```

```
    //
    Method(STC, 2)
    {
        //
        // This sensor only supports a lower critical threshold.
        //
        If(And(Arg0, 0x08))
        {
            //
            // If this threshold should be disable write the value
            // 0 (zero) to the threshold register.  Otherwise convert
            // the value from RPM to 'count' and store.
            //
            If(LEqual(Arg1, 0))
            {
                Store(0, Local0)
            }
            Else
            {
                If(LEqual(Arg1, 0x7FFFFFFF))
                {
                    Store(0, Local0)
                }
                Else
                {
                    Divide(675000, Arg1, Local1, Local0)
                }
            }

            \_SB.SMB0._SBW(0x06, 0x2C, 0x3C, 0x01, Local0, 0x00)
        }
    } // STC()

} // Device(FS01)

////////////////////////////////////////////////////////////////////////////
// Device:      VS00
// ------------------------------------------------------------------------
// Description: +12V Voltage Sensor on Heceta II.
////////////////////////////////////////////////////////////////////////////
Device(VS00)
{
    Name(_HID, "MGMT103")
    Name(_UID, 0x01030100)                  // Sensor Signature (1st Voltage Sensor)

    //
    // INIT (Sensor Initialization):
    // -----------------------------
    // Initialize this sensor.
    //
    Method(INIT)
    {
        //
        // Configure the upper and lower critical thresholds to their default
        // values (+/- 1.2V of nominal).
        //
        STC(0x04, 13200)
        STC(0x08, 10800)
    } // INIT()

    //
    // INFO (Sensor Information):
    // --------------------------
    // Return a SENSOR_INFO structure describing the properties of this sensor.
    //
    Method(INFO)
    {
        Return(Buffer() {
            0x01,                           // Object Type (Sensor)
            0x10,                           // Version (v1.0)
            0x03,                           // Major Sensor Type (Power Quality)
```

```
            0x01,                           // Minor Sensor Type (Standard Voltage Sensor)
            0x00,                           // Sensor Instance (1st Instance)
            0x07,                           // Hardware Capability (State-based Numeric
                                            //              sensor w/ Thresholds)
            0x30,                           // Monitored Device Type (Power Source)
            0x00,                           // Monitored Device Instance (1st Instance)
            0x09,                           // Property Count
            0x01, 0x00, 0x00, 0x2E, 0xE0,   // Property: Nominal Reading (12.0 volts)
            0x02, 0x00, 0x00, 0x3E, 0x42,   // Property: Physical Maximum Reading (15.938 volts)
            0x03, 0x00, 0x00, 0x00, 0x3F,   // Property: Physical Minimum Reading (0.063 volts)
            0x05, 0x00, 0x00, 0x00, 0x3F,   // Property: Resolution (0.063 volts)
            0x10, 0x00, 0x00, 0x00, 0x05,   // Property: State Capability (OK & Critical States)
            0x11, 0x00, 0x00, 0x01, 0x2C,   // Property: Polling Frequency (5 minutes)
            0x20, 0x00, 0x00, 0x00, 0x0C,   // Property: Threshold Capability (Upper &
                                            //                   Lower Critical Thresholds)
            0x23, 0x00, 0x00, 0x33, 0x90,   // Property: Default Upper Critical
                                            //                     Threshold (13.2 volts)
            0x24, 0x00, 0x00, 0x2A, 0x30,   // Property: Default Lower Critical
                                            //                     Threshold (10.8 volts)
        })
    } // INFO()

    //
    // Sensor Reading (SR):
    // --------------------
    // Return the current sensor reading.
    //
    Method(SR)
    {
        //
        // Set the return value to UNKNOWN.
        //
        Store(0x80000000, Local0)

        //
        // Read the voltage and make sure the data is valid.
        //
        Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x24), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 2)), Local0)

            //
            // For the +12V input, the voltage can be calculated by
            // multiplying the current reading by 0.063 volts (the
            // resolution).
            //
            Multiply(Local0, 63, Local0)
        }

        Return (Local0)
    } // SR()

    //
    // Sensor State (SS):
    // ------------------
    // Heceta II ASICs include status registers but these cannot be used
    // due to the automatic clearing of these register after any read
    // operation.  Instead, we must get the sensor reading and manually
    // compare against the current thresholds.
    //
    Method(SS)
    {
        //
        // Get the current upper and lower critical thresholds.  If both
        // are disabled, return the status as OK.  If both are unknown,
        // return the status as UNKNOWN.
        //
        Store(STQ(0x04), Local1)
        Store(STQ(0x08), Local2)
```

```
        If(LEqual(Local1, 0x7FFFFFFF))
        {
            If(LEqual(Local2, 0x7FFFFFFF))
            {
                Return(0x01)        // OK state.
            }
        }

        If(LEqual(Local1, 0x80000000))
        {
            If(LEqual(Local2, 0x80000000))
            {
                Return(0x80000000)  // UNKNOWN state.
            }
        }

        //
        // Get the current reading and make sure the data is valid.
        //
        Store(SR, Local3)
        If(LEqual(Local3, 0x80000000))
        {
            Return(0x80000000)  // UNKNOWN state.
        }

        //
        // If the upper critical threshold is valid, compare
        // against the current reading.
        //
        If(LLess(Local1, 0x7FFFFFFF))
        {
            If(LGreaterEqual(Local3, Local1))
            {
                Return(0x04)        // CRITICAL state.
            }
        }

        //
        // If the lower critical threshold is valid, compare
        // against the current reading.
        //
        If(LLess(Local2, 0x7FFFFFFF))
        {
            If(LLessEqual(Local3, Local2))
            {
                Return(0x04)        // CRITICAL state.
            }
        }

        Return(0x01)                // OK state.
    } // SS()

    //
    // Sensor Threshold Query (STQ):
    // -----------------------------
    // Returns current threshold values.
    //
    Method(STQ, 1)
    {
        //
        // Set the return value to UNKNOWN.
        //
        Store(0x80000000, Local0)

        //
        // Upper Critical Threshold?
        //
        If(And(Arg0, 0x04))
        {
            //
            // Get the current upper critical threshold value and make
```

```
            // sure the data is valid.  Note that the value 0xFF indicates
            // that the threshold is disabled.
            //
            Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x33), Local1)
            If(LEqual(Derefof(Index(Local1, 0)), 0))
            {
                Store(Derefof(Index(Local1, 2)), Local0)

                If(LEqual(Local0, 0xFF))
                {
                    Store(0x7FFFFFFF, Local0)
                }
                Else
                {
                    //
                    // Convert reading to millivolts.
                    //
                    Multiply(Local0, 63, Local0)
                }
            }
        }

        //
        // Lower Critical Threshold?
        //
        If(And(Arg0, 0x08))
        {
            //
            // Get the current lower critical threshold value and make
            // sure the data is valid.  Note that the value 0x00 indicates
            // that the threshold is disabled.
            //
            Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x34), Local1)
            If(LEqual(Derefof(Index(Local1, 0)), 0))
            {
                Store(Derefof(Index(Local1, 2)), Local0)

                If(LEqual(Local0, 0x00))
                {
                    Store(0x7FFFFFFF, Local0)
                }
                Else
                {
                    //
                    // Convert reading to millivolts.
                    //
                    Multiply(Local0, 63, Local0)
                }
            }
        }

        Return(Local0)
    } // STQ()

    //
    // Sensor Threshold Control (STC):
    // -------------------------------
    // Updates threshold values.
    //
    Method(STC, 2)
    {
        //
        // Upper Critical Threshold?
        //
        If(And(Arg0, 0x04))
        {
            //
            // Check the new threshold value.  If this threshold should
            // be disable write the value 0xFF to the limit register.
            //
            If(LEqual(Arg1, 0x7FFFFFFF))
```

```
            {
                Store(0xFF, Local0)
            }
            Else
            {
                If(LNotEqual(Arg1, 0))
                {
                    Divide(Arg1, 63, Local1, Local0)
                }
            }

            \_SB.SMB0._SBW(0x06, 0x2C, 0x33, 0x01, Local0, 0x00)
        }

        //
        // Lower Critical Threshold?
        //
        If(And(Arg0, 0x08))
        {
            //
            // Check the new threshold value.  If this threshold should
            // be disable write the value 0x00 to the limit register.
            //
            If(LEqual(Arg1, 0x7FFFFFFF))
            {
                Store(0x00, Local0)
            }
            Else
            {
                If(LNotEqual(Arg1, 0))
                {
                    Divide(Arg1, 63, Local1, Local0)
                }
            }

            \_SB.SMB0._SBW(0x06, 0x2C, 0x34, 0x01, Local0, 0x00)
        }

    } // STC()

} // Device(VS00)

////////////////////////////////////////////////////////////////////////////
// Device:      VS01
// ------------------------------------------------------------------------
// Description: +5V Voltage Sensor on Heceta II.
////////////////////////////////////////////////////////////////////////////
Device(VS01)
{
    Name(_HID, "MGMT103")
    Name(_UID, 0x01030101)                  // Sensor Signature (2nd Voltage Sensor)

    //
    // INIT (Sensor Initialization):
    // -----------------------------
    // Initialize this sensor.
    //
    Method(INIT)
    {
        //
        // Configure the upper and lower critical thresholds to their default
        // values (6.0V and 4.0V, respectively).
        //
        STC(0x04, 6000)
        STC(0x08, 4000)
    } // INIT()

    //
    // INFO (Sensor Information):
    // --------------------------
    // Return a SENSOR_INFO structure describing the properties of this sensor.
```

```
//
Method(INFO)
{
    Return(Buffer() {
        0x01,                           // Object Type (Sensor)
        0x10,                           // Version (v1.0)
        0x03,                           // Major Sensor Type (Power Quality)
        0x01,                           // Minor Sensor Type (Standard Voltage Sensor)
        0x01,                           // Sensor Instance (2nd Instance)
        0x07,                           // Hardware Capability (State-based Numeric
                                        //                sensor w/ Thresholds)
        0x30,                           // Monitored Device Type (Power Source)
        0x00,                           // Monitored Device Instance (1st Instance)
        0x09,                           // Property Count
        0x01, 0x00, 0x00, 0x13, 0x88,   // Property: Nominal Reading (5.0 volts)
        0x02, 0x00, 0x00, 0x19, 0xF0,   // Property: Physical Maximum Reading (6.640 volts)
        0x03, 0x00, 0x00, 0x00, 0x16,   // Property: Physical Minimum Reading (0.026 volts)
        0x05, 0x00, 0x00, 0x00, 0x16,   // Property: Resolution (0.026 volts)
        0x10, 0x00, 0x00, 0x00, 0x05,   // Property: State Capability (OK & Critical States)
        0x11, 0x00, 0x00, 0x01, 0x2C,   // Property: Polling Frequency (5 minutes)
        0x20, 0x00, 0x00, 0x00, 0x0C,   // Property: Threshold Capability (Upper &
                                        //                      Lower Critical Threshold)
        0x23, 0x00, 0x00, 0x17, 0x70,   // Property: Default Upper Critical
                                        //                       Threshold (6.0 volts)
        0x24, 0x00, 0x00, 0x0F, 0xA0,   // Property: Default Lower Critical
                                        //                       Threshold (4.0 volts)
    })
} // INFO()

//
// Sensor Reading (SR):
// --------------------
// Return the current sensor reading.
//
Method(SR)
{
    //
    // Set the return value to UNKNOWN.
    //
    Store(0x80000000, Local0)

    //
    // Read the voltage and make sure the data is valid.
    //
    Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x23), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
        Store(Derefof(Index(Local1, 2)), Local0)

        //
        // For the +5V input, the voltage can be calculated by
        // multiplying the current reading by 0.026 volts (the
        // resolution).
        //
        Multiply(Local0, 26, Local0)
    }

    Return (Local0)
} // SR()

//
// Sensor State (SS):
// ------------------
// Heceta II ASICs include status registers but these cannot be used
// due to the automatic clearing of these register after any read
// operation.  Instead, we must get the sensor reading and manually
// compare against the current thresholds.
//
Method(SS)
{
    //
```

```
    // Get the current upper and lower critical thresholds.  If both
    // are disabled, return the status as OK.  If both are unknown,
    // return the status as UNKNOWN.
    //
    Store(STQ(0x04), Local1)
    Store(STQ(0x08), Local2)

    If(LEqual(Local1, 0x7FFFFFFF))
    {
        If(LEqual(Local2, 0x7FFFFFFF))
        {
            Return(0x01)        // OK state.
        }
    }

    If(LEqual(Local1, 0x80000000))
    {
        If(LEqual(Local2, 0x80000000))
        {
            Return(0x80000000)  // UNKNOWN state.
        }
    }

    //
    // Get the current reading and make sure the data is valid.
    //
    Store(SR, Local3)
    If(LEqual(Local3, 0x80000000))
    {
        Return(0x80000000)  // UNKNOWN state.
    }

    //
    // If the upper critical threshold is valid, compare
    // against the current reading.
    //
    If(LLess(Local1, 0x7FFFFFFF))
    {
        If(LGreaterEqual(Local3, Local1))
        {
            Return(0x04)        // CRITICAL state.
        }
    }

    //
    // If the lower critical threshold is valid, compare
    // against the current reading.
    //
    If(LLess(Local2, 0x7FFFFFFF))
    {
        If(LLessEqual(Local3, Local2))
        {
            Return(0x04)        // CRITICAL state.
        }
    }

    Return(0x01)                // OK state.
} // SS()

//
// Sensor Threshold Query (STQ):
// -----------------------------
// Returns current threshold values.
//
Method(STQ, 1)
{
    //
    // Set the return value to UNKNOWN.
    //
    Store(0x80000000, Local0)
```

```
        //
        // Upper Critical Threshold?
        //
        If(And(Arg0, 0x04))
        {
            //
            // Get the current upper critical threshold value and make
            // sure the data is valid.  Note that the value 0xFF indicates
            // that the threshold is disabled.
            //
            Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x31), Local1)
            If(LEqual(Derefof(Index(Local1, 0)), 0))
            {
                Store(Derefof(Index(Local1, 2)), Local0)

                If(LEqual(Local0, 0xFF))
                {
                    Store(0x7FFFFFFF, Local0)
                }
                Else
                {
                    //
                    // Convert reading to millivolts.
                    //
                    Multiply(Local0, 26, Local0)
                }
            }
        }

        //
        // Lower Critical Threshold?
        //
        If(And(Arg0, 0x08))
        {
            //
            // Get the current lower critical threshold value and make
            // sure the data is valid.  Note that the value 0x00 indicates
            // that the threshold is disabled.
            //
            Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x32), Local1)
            If(LEqual(Derefof(Index(Local1, 0)), 0))
            {
                Store(Derefof(Index(Local1, 2)), Local0)

                If(LEqual(Local0, 0x00))
                {
                    Store(0x7FFFFFFF, Local0)
                }
                Else
                {
                    //
                    // Convert reading to millivolts.
                    //
                    Multiply(Local0, 26, Local0)
                }
            }
        }

        Return(Local0)
    } // STQ()

    //
    // Sensor Threshold Control (STC):
    // -------------------------------
    // Updates threshold values.
    //
    Method(STC, 2)
    {
        //
        // Upper Critical Threshold?
        //
```

```
        If(And(Arg0, 0x04))
        {
            //
            // Check the new threshold value.  If this threshold should
            // be disable write the value 0xFF to the limit register.
            //
            If(LEqual(Arg1, 0x7FFFFFFF))
            {
                Store(0xFF, Local0)
            }
            Else
            {
                If(LNotEqual(Arg1, 0))
                {
                    Divide(Arg1, 26, Local1, Local0)
                }
            }

            \_SB.SMB0._SBW(0x06, 0x2C, 0x31, 0x01, Local0, 0x00)
        }

        //
        // Lower Critical Threshold?
        //
        If(And(Arg0, 0x08))
        {
            //
            // Check the new threshold value.  If this threshold should
            // be disable write the value 0x00 to the limit register.
            //
            If(LEqual(Arg1, 0x7FFFFFFF))
            {
                Store(0x00, Local0)
            }
            Else
            {
                If(LNotEqual(Arg1, 0))
                {
                    Divide(Arg1, 26, Local1, Local0)
                }
            }

            \_SB.SMB0._SBW(0x06, 0x2C, 0x32, 0x01, Local0, 0x00)
        }

    } // STC()

} // Device(VS01)

////////////////////////////////////////////////////////////////////////////
// Device:      VS02
// ----------------------------------------------------------------------
// Description: TODO
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Device:      VS03
// ----------------------------------------------------------------------
// Description: TODO
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Device:      VS04
// ----------------------------------------------------------------------
// Description: TODO
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Device:      VS05
// ----------------------------------------------------------------------
// Description: TODO
```

```
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Device:      IS00
// ------------------------------------------------------------------------
// Description: Chassis Intrusion Sensor on Heceta II.
////////////////////////////////////////////////////////////////////////////
Device(IS00)
{
    Name(_HID, "MGMT104")
    Name(_UID, 0x01040100)                  // Sensor Signature (1st Chassis Intrusion Sensor)

    //
    // INIT (Sensor Initialization):
    // -----------------------------
    // Initialize this sensor.
    //
    Method(INIT)
    {
        // Nothing required for this sensor...
    } // INIT()

    //
    // INFO (Sensor Information):
    // --------------------------
    // Return a SENSOR_INFO structure describing the properties of this sensor.
    //
    Method(INFO)
    {
        Return(Buffer() {
            0x01,                           // Object Type (Sensor)
            0x10,                           // Version (v1.0)
            0x04,                           // Major Sensor Type (Physical Security)
            0x01,                           // Minor Sensor Type (Standard Chassis
                                            //                    Intrusion Sensor)
            0x00,                           // Sensor Instance (1st Instance)
            0x42,                           // Hardware Capability (State-based Discrete
                                            //                      w/ Manual Re-arm)
            0x00,                           // Monitored Device Type (System Chassis)
            0x00,                           // Monitored Device Instance (1st Instance)
            0x02,                           // Property Count
            0x10, 0x00, 0x00, 0x00, 0x03,   // Property: State Capability (OK & Warning States)
            0x11, 0x00, 0x00, 0x01, 0x2C,   // Property: Polling Frequency (5 minutes)
        })
    } // INFO()

    //
    // Sensor State (SS):
    // ------------------
    // Read the current chassis intrusion status from the second Interrupt
    // Status Register.  Note that this register gets cleared after each
    // read, and isn't repopulated with data until after the next Heceta II
    // duty cycle.
    //
    Method(SS)
    {
        //
        // Set the return value to UNKNOWN.
        //
        Store(0x80000000, Local0)

        //
        // Read the second status register to see if a chassis
        // intrusion has occurred.
        //
        Store(\_SB.SMB0._SBR(0x07, 0x2C, 0x42), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 2)), Local0)

            If(And(0x10, Local0))
```

```
            {
                Store(0x02, Local0)   // WARNING state.
            }
            Else
            {
                Store(0x01, Local0)   // OK state.
            }
        }

        Return(Local0)
    } // SS()

    //
    // Sensor Re-Arm (SRA):
    // --------------------
    // Re-arm a manually arming sensor.
    //
    Method(SRA)
    {
        //
        // Bit 7 of the Chassis Intrusion Clear register causes a re-arm.
        //
        \_SB.SMB0._SBW(0x06, 0x2C, 0x46, 0x01, 0x80, 0x00)
    } // SS()

} // Device(IS00)
```

### A.2.4 Alert on LAN

The Alert on LAN ASIC includes watchdog hardware that is modeled in this sample ASL as an OS hang watchdog.

For simplicity, the watchdog object in this example has been added to the root of the system bus (_SB) hierarchy in the ACPI namespace. Ideally this object should exist under the bus that it is connected to (e.g. SMB0) – as shown in Figure 1.

For this example we'll assume that the Alert on LAN ASIC is connected to the PIIX4 SMBus at slave address 0x2E (7-bit non-shifted value). A full listing of the ASL to implement this functionality is provided below.

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//                      Alert on LAN* Watchdog Definition                 //
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
// Device:      OW00
// ------------------------------------------------------------------------
// Description: OS Hang Watchdog on Alert on LAN (AOL) ASIC.
////////////////////////////////////////////////////////////////////////////
Device(OW00)
{
    Name(_HID, "MGMT201")
    Name(_UID, 0x02010400)                  // Watchdog Signature (1st OS Hang Watchdog)

    Name(WTOD, 0xD7)                        // Keeps track of the current watchdog timeout
duration.

    //
    // INIT (Sensor Initialization):
    // -----------------------------
    // Initialize this watchdog.
    //
    Method(INIT)
    {
        //
        // Initialize the watchdog timeout duration to the default value
        // (215 seconds).
        //
        WTC(\_SB.OW00.WTOD)
    } // INIT()

    //
    // INFO (Watchdog Information):
    // ----------------------------
    // Return a WATCHDOG_INFO structure describing the properties of this watchdog.
    //
    Method(INFO)
    {
        Return(Buffer() {
            0x02,                           // Object Type (Watchdog)
            0x10,                           // Version (v1.0)
            0x01,                           // Major Watchdog Type (System Hang)
            0x04,                           // Minor Watchdog Type (Standard OS Hang Watchdog)
            0x00,                           // Watchdog Instance (1st Instance)
            0x04,                           // Property Count
            0x01, 0x00, 0x00, 0x00, 0x01,   // Property: Action Capability (Generate
                                            //                                 Remote Alert)
            0x02, 0x00, 0x00, 0x00, 0x2B,   // Property: Timer Resolution (43 seconds)
            0x03, 0x00, 0x00, 0x00, 0xD7,   // Property: Default Timeout (215 seconds)
            0x04, 0x00, 0x00, 0x00, 0x01    // Property: Default Action (Generate Remote Alert)
        })
    } // INFO()

    //
    // WTV (Watchdog Timer Value):
```

```
// --------------------------
// Returns the current value of the watchdog timer.
//
Method(WTV)
{
    //
    // Set the return value to UNKNOWN.
    //
    Store(0x80000000, Local0)

    //
    // Read the current timer value from the "Watchdog Timer" register.
    // Note that the timeout is specified in 43-second units using
    // bits 1 through 7.
    //
    Store(\_SB.SMB0._SBR(0x07, 0x2E, 0x06), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
        Store(Derefof(Index(Local1, 2)), Local0)
        ShiftRight(Local0, 1, Local0)
        Multiply(Local0, 43, Local0)
    }

    Return(Local0)
} // WTV()

//
// WTR (Watchdog Timer Reset):
// --------------------------
// Reset the watchdog timer to current timeout duration.
//
Method(WTR)
{
    //
    // Reset the watchdog timer by simply reading and re-writing
    // the "Watchdog Timer" register.  Note that this does not
    // enable a disabled watchdog (see WAC).
    //
    Store(\_SB.SMB0._SBR(0x07, 0x2E, 0x06), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
        Store(Derefof(Index(Local1, 2)), Local0)
        \_SB.SMB0._SBW(0x06, 0x2E, 0x06, 0x01, Local0, 0x00)
    }
} // WTR()

//
// WTQ (Watchdog Timeout Query):
// ----------------------------
// Returns the value of the current watchdog timeout duration.
//
Method(WTQ)
{
    Return(\_SB.OW00.WTOD)
} // WTQ()

//
// WTC (Watchdog Timeout Control):
// ------------------------------
// Sets the value of the watchdog timeout duration.
//
Method(WTC, 1)
{
    //
    // Preserve bit 0 (Watchdog Enable) of the "Watchdog Timer"
    // register, disable the watchdog (required to update the
    // timeout duration), and write the new timeout duration.
    //
    Store(\_SB.SMB0._SBR(0x07, 0x2E, 0x06), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
```

```
            Store(Derefof(Index(Local1, 2)), Local2)
            And(Local2, 0x01, Local3)       // Preserve bit 0.

            If(Local3)
            {
                And(Local2, 0xFE, Local2)
                \_SB.SMB0._SBW(0x06, 0x2E, 0x06, 0x01, Local2, 0x00)
            }

            //
            // Compute the new timeout duration as the number of 43-second
            // units, rounding the result to the nearest unit.  (Note that
            // the new timeout duration must equal one or more units.)
            //
            If(LEqual(Arg0, 0))
            {
                Store(2, Local0)            // Default to one unit (bits 7:1).
            }
            Else
            {
                Divide(Arg0, 43, Local1, Local0)

                If(LGreater(Local1, 21))
                {
                    Add(Local0, 1, Local0)
                }
                Else
                {
                    If(LEqual(Local0, 0))
                    {
                        Store(1, Local0)
                    }
                }
                ShiftLeft(Local0, 1, Local0)
            }

            //
            // Configure bit 0 (Watchdog Enable) appropriately and
            // update the "Watchdog Timer" register.
            //
            If(Local3)
            {
                Add(Local0, 1, Local0)
            }

            \_SB.SMB0._SBW(0x06, 0x2E, 0x06, 0x01, Local0, 0x00)

            //
            // Store this new timeout duration for later access...
            //
            Store(Arg0,\_SB.OW00.WTOD)
        }
    } // WTC()

    //
    // WAQ (Watchdog Action Query):
    // ----------------------------
    // Returns a bitmap representing the actions that this watchdog will
    // perform when a timeout occurs.  Note that the only action supported
    // by this watchdog hardware is to send a remote alert.
    //
    Method(WAQ)
    {
        //
        // Set the return value to UNKNOWN.
        //
        Store(0x80000000, Local0)

        //
        // Read bit 6 of the AOL "SOS Event Status Mask" register to see
        // if this watchdog is enabled for remote alerts.
```

```
    //
    Store(\_SB.SMB0._SBR(0x07, 0x2E, 0x03), Local1)
    If(LEqual(Derefof(Index(Local1, 0)), 0))
    {
        Store(Derefof(Index(Local1, 2)), Local0)

        If(And(0x40, Local0))
        {
           Store(1, Local0)        // Remote Alerting Enabled.
        }
        Else
        {
            Store(0, Local0)        // No Actions Enabled.
        }
    }

    Return(Local0)
} // WAQ()

//
// WAC (Watchdog Action Control):
// ------------------------------
// Sets the action(s) to be performed by the watchdog hardware
// when a timeout occurs.  Note that the only action supported
// by this watchdog hardware is to send a remote alert.  A value
// of zero (no action) disables this watchdog.
//
Method(WAC, 1)
{
    //
    // If the watchdog should be disabled, get the current timeout
    // (to preserve) from the "Watchdog Timer" register and clear
    // bit 0 (watchdog enable).  Otherwise, compute the new timeout
    // duration, make sure the hardware is enabled, and write.
    //
    If(LEqual(Arg0, 0))
    {
        //
        // Preserve the current timeout duration, but clear bit 0
        // to ensure the watchdog is disabled. If we can't read the
        // current timeout duration store the default (5).
        //
        Store(\_SB.SMB0._SBR(0x07, 0x2E, 0x06), Local1)
        If(LEqual(Derefof(Index(Local1, 0)), 0))
        {
            Store(Derefof(Index(Local1, 2)), Local0)
            And(Local0, 0xFE, Local0)
        }
        Else
        {
            Store(0x0A, Local0)
        }

        \_SB.SMB0._SBW(0x06, 0x2E, 0x06, 0x01, Local0, 0x00)
    }
    Else
    {
        //
        // Enable Remote Alerts?
        //
        If(And(Arg0, 1))
        {
            //
            // Enable remote alerting by setting bit 6 of the
            // "SOS Event Status Mask" register.  Preserve
            // all other bits.
            //
            Store(\_SB.SMB0._SBR(0x07, 0x2E, 0x03), Local1)
            If(LEqual(Derefof(Index(Local1, 0)), 0))
            {
                Store(Derefof(Index(Local1, 2)), Local0)
```

```
                    Or(Local0, 0x40, Local0)
                    \_SB.SMB0._SBW(0x06, 0x2E, 0x03, 0x01, Local0, 0x00)
                }

                //
                // Make sure the watchdog hardware is enabled by
                // ensuring bit 0 of the "Watchdog Timer" register
                // is enabled.
                //
                Store(\_SB.SMB0._SBR(0x07, 0x2E, 0x06), Local1)
                If(LEqual(Derefof(Index(Local1, 0)), 0))
                {
                    Store(Derefof(Index(Local1, 2)), Local0)
                    Or(Local0, 0x01, Local0)
                    \_SB.SMB0._SBW(0x06, 0x2E, 0x06, 0x01, Local0, 0x00)
                }
            }
            Else
            {
                //
                // Disable remote alerting by clearing bit 6 of the
                // "SOS Event Status Mask" register.  Preserve all
                // other bits.
                //
                Store(\_SB.SMB0._SBR(0x07, 0x2E, 0x03), Local1)
                If(LEqual(Derefof(Index(Local1, 0)), 0))
                {
                    Store(Derefof(Index(Local1, 2)), Local0)
                    And(Local0, 0x3F, Local0)
                    \_SB.SMB0._SBW(0x06, 0x2E, 0x03, 0x01, Local0, 0x00)
                }
            }
        }

    } // WAC()

} // Device(OW00)
```